Performance Characteristics of a Kernel-Space Packet Capture Module

THESIS

Samuel W. Birch, IA-04, DAF

AFIT/GCO/ENG/10-03

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCO/ENG/10-03

Performance Characteristics of a Kernel-Space Packet Capture Module

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Samuel W. Birch, B.S.

IA-04, DAF

March 2010

AFIT/GCO/ENG/10-03

Performance Characteristics of a Kernel-Space Packet Capture Module

Samuel W. Birch, B.S.
IA-04, DAF

Approved:

Robert F Mills	2 Mar 10
Dr. Robert F. Mills, PhD (Chairman)	date
Barry Mullins	2 Mar 10
Dr. Barry E. Mullins (Member)	date
Kenneth M Hopkinson	2 Mar 10
Dr. Kenneth M. Hopkinson (Member)	date

Abstract

Defending networks, network-connected assets, and the information they both carry and store is an operational challenge and a significant drain on resources. A plethora of historical and ongoing research efforts are focused on increasing the effectiveness of the defenses or reducing the costs of existing defenses. One valuable facet in defense is the ability to perform *post mortem* analysis of incidents that have occurred, and this tactic requires accurate storage and rapid retrieval of vast quantities of historical network data.

This research improves the efficiency of capturing network packets to disk using commodity, general-purpose hardware and operating systems. It examines the bottlenecks between Network Interface Card (NIC) and disk, implements a kernel-space capture capability to improve storage efficiency, and analyzes the performance characteristics of this approach. The proof of concept PKAP kernel-space packet capture module avoids the penalties associated with both the kernel-to-user and user-to-kernel space memory copies, removing unnecessary overhead and improving the ability of a network capture system to accurately capture higher network rates with lower computational overhead.

Results show that a kernel-space NIC-to-Disk (N^2D) is both possible and beneficial. The PKAP kernel module can capture packets to disk with a packet drop rate 8.9% less than the user-space equivalent, at a 95% confidence interval. During the high levels of disk I/O contention produced by queries for the captured data, the PKAP implementation shows a 3% reduction in CPU utilization, and overall the PKAP implementation reduces memory utilization of the capture process by 16%.

Acknowledgments

First and foremost, I thank God for His strength and grace throughout this process. I am reminded daily of how little I control and how much I depend on the Lord.

I thank my wife and children for their love, support, patience, and understanding during the last twenty-two months: This work is as much yours as it is mine—even if this thesis is "the kind of book that you get bored with after two seconds of reading it."

I thank my thesis adviser, Dr. Robert Mills, for providing the right balance of freedom to explore my ideas, guidance to keep me on track, and patience to bear with my rabbit trails. I also thank Dr. Barry Mullins and Dr. Ken Hopkinson for their support and assistance.

Samuel W. Birch

Contents

Page

Abstract iv

Acknowledgments v

List of Figures x

List of Tables xi

List of Acronyms xii

I. Introduction 1
- 1.1 Motivation 1
- 1.2 Problem 2
- 1.3 Research Objectives 3
- 1.4 Limitations 4
- 1.5 Implications 5
- 1.6 Thesis Structure 6

II. Literature Review 7
- 2.1 Background On Packet Capture 7
  - 2.1.1 Sniffing 7
  - 2.1.2 Capture Types 8
    - 2.1.2.1 Live Analysis 8
    - 2.1.2.2 Cyber Hind-Sight 8
  - 2.1.3 Capture Tools 9
    - 2.1.3.1 Common Tools 9
    - 2.1.3.2 High-End Capture Tools 11
- 2.2 NIC-to-Disk Packet Capture and Network Security . 12
  - 2.2.1 Purpose 12
  - 2.2.2 Execution 13
    - 2.2.2.1 Basic Architecture 13
    - 2.2.2.2 Extended Architecture 15
  - 2.2.3 Cost 16
  - 2.2.4 Technical Challenges 17
- 2.3 Packet Capture Mechanics 18
  - 2.3.1 Overview 18
  - 2.3.2 AF_PACKET (libpcap-1.0.0 vanilla) 19

Page

2.3.2.1 Packet Path . . . 19
2.3.2.2 The *pcap* File Format . . . 21
2.3.3 NAPI . . . 23
2.3.4 libpcap-mmap . . . 25
2.3.5 PF_RING: libpcap-ring . . . 26
2.3.5.1 The PF_RING Kernel Module . . . 28
2.3.5.2 The pfring library . . . 31
2.3.6 DMA Ring Module . . . 34
2.4 Storage Efficiency . . . 34
2.5 Summary . . . 34

III. Design . . . 36
3.1 Problem Definition . . . 36
3.1.1 Goals and Hypothesis . . . 36
3.1.2 Approach . . . 38
3.1.2.1 Solution Space . . . 38
3.1.2.2 Direction . . . 38
3.1.2.3 Prototype . . . 39
3.2 Moving to the Kernel . . . 40
3.2.1 Linux Kernel Modules . . . 40
3.2.2 Linux Kernel Threads . . . 41
3.2.3 Capturing Packets from Kernel-Space . . . 43
3.2.3.1 Selection of PF_RING . . . 43
3.2.3.2 Modifying PF_RING . . . 44
3.2.3.3 Accessing PF_RING from PKAP . . . 46
3.2.4 Breaking the Rules (Writing Files from Kernel-Space) . . . 48
3.3 Architecture Overview . . . 50

IV. Methodology . . . 52
4.1 System Boundaries . . . 52
4.2 Evaluation Methodology . . . 54
4.3 System Services . . . 55
4.4 System Workload . . . 56
4.4.1 Network Traffic . . . 57
4.4.1.1 Characteristics . . . 57
4.4.1.2 Bitrates . . . 59
4.4.1.3 Duration . . . 59
4.4.2 Data Queries . . . 59

Page

4.4.2.1 Query Object Creation 60
4.4.2.2 Query-Response Workload 60
4.4.3 Disk Maintenance 60
4.5 System Performance Metrics 61
4.6 System Parameters 61
4.6.1 Capture Method 62
4.6.2 Disk Device Selection 62
4.6.3 Filtering 62
4.6.4 System Specifications 63
4.7 System Factors 63
4.7.1 Capture Method 63
4.7.1.1 PKAP Kernel Module 64
4.7.1.2 DaemonLogger 64
4.7.2 Storage Device 65
4.7.3 Network Workload 66
4.7.3.1 Packet Size 66
4.7.3.2 Bitrate 67
4.7.4 Query Workload 68
4.8 Evaluation Technique and Environment 69
4.8.1 Evaluation Environment 69
4.8.2 Evaluation Technique 71
4.8.2.1 Dropped Packet Rate 71
4.8.2.2 CPU Utilization 72
4.8.2.3 Memory Utilization 72
4.8.2.4 Query Delay 73
4.9 Experiment Design 73
4.9.1 Experiment 0: Pilot Testing 74
4.9.2 Experiment 1: Packets to Disk 75
4.9.3 Experiment 2: Packets to `/dev/null` 75
4.9.4 Experiment 3: Impact of Data Queries 76
4.9.5 Methodology Summary 77

V. Analysis 78
5.1 Results and Analysis of Experiment 1 78
5.1.1 Experiment 1 Overview 78
5.1.2 Packet Drop Rate 80
5.1.3 CPU Utilization 83
5.1.4 Memory Utilization 83

Page

5.1.5 Summary Analysis 85
5.2 Results and Analysis of Experiment 2 87
5.2.1 Experiment 2 Overview 87
5.2.2 Packet Drop Rate 90
5.2.3 CPU Utilization 91
5.2.4 Memory 92
5.2.5 Summary Analysis 92
5.3 Results and Analysis of Experiment 3 93
5.3.1 Experiment 3 Overview 93
5.3.2 Query Metrics 93
5.3.3 Packet Drop Rate 97
5.3.4 CPU Utilization 98
5.3.5 Memory 99
5.3.6 Summary Analysis 99
5.4 Overall Analysis 99

VI. Discussion 101
6.1 Conclusions 101
6.1.1 Construct a Kernel-space N_{Q}^{2} Capability . . . 101
6.1.2 Reduce Packet Drop Rate of the Capture Process 102
6.1.3 Reduce CPU Utilization of the Capture Process 102
6.1.4 Reduce Memory Utilization of the Capture Process 103
6.1.5 Retain Query Response Performance of the Capture Process 104
6.2 Significance 104
6.3 Future Efforts 105

Appendix A. DaemonLogger Modifications 106

Appendix B. PF_RING Modifications 108

Bibliography 114

Vita 117

List of Figures

Figure		Page
2.1	Simple NIC-to-Disk Capture System Architecture	13
2.2	The Extended Architecture of a NIC-to-Disk Capture System	16
2.3	The libpcap-1.0.0 (mainstream) Capture Process	19
2.4	The Libpcap File Format	24
2.5	The lipcap-1.0.0 + NAPI Capture Process	25
2.6	The libpcap-mmap + NAPI Capture Process	26
2.7	The libpcap-ring + NAPI Capture Process	27
2.8	PF_RING-Aware NIC Driver Flow	30
2.9	PF_RING Protocol Handler Basics	31
2.10	The libpcap-DMA-ring Capture Process	33
3.1	The PKAP Capture Process	50
4.1	Systems Approach Diagram for the NIC-to-Disk Capture System	53
4.2	Diagram of the Evaluation Environment	69
5.1	Experiment 1: Summary Graphs of Key Metrics	79
5.2	Experiment 1: Packet Size Impact	81
5.3	Experiment 1: KBps Written / Bitrate Level	82
5.4	Experiment 1: Relationship Between Dropped Packets and CPU Utilization	86
5.5	Experiment 2: Summary Graphs of Key Metrics	88
5.6	Experiment 1: Packet Size Impact	89
5.7	Experiment 3: Summary Graphs of Key Process-Based Metrics	94
5.8	Experiment 3: Summary Graphs of Key Query-Based Metrics	95

List of Tables

Table		Page
4.1	Table of Factor Configuration Sets	73
5.1	Experiment 1 Hypothesis Testing of Dropped Packet Rate .	82
5.2	Experiment 1 Hypothesis Testing of CPU Utilization	84
5.3	Experiment 2 Hypothesis Testing of Dropped Packet Rate .	91
5.4	Experiment 2 Hypothesis Testing of CPU Utilization	92
5.5	Experiment 3 Hypothesis Testing of Query Performance . .	97
5.6	Experiment 3 Hypothesis Testing of Dropped Packet Rate .	98
5.7	Experiment 3 Hypothesis Testing of CPU Utilization	98

List of Acronyms

Acronym		Page
ACL	Access Control List	57
API	Application Programming Interface	14
ASIC	Application Specific Integrated Circuit	11
BPF	Berkeley Packet Filter	4
BSD	Berkeley Software Distribution	18
caplen	capture length	32
CERT	Computer Emergency Response Team	16
CPU	Central Processing Unit	4
CUT	Component Under Test	52
DMA	Direct Memory Access	11
EOF	End Of File	65
eSATA	External Serial Advanced Technology Attachement	71
ext4	Extended Filesystem version 4	62
FDDI	Fiber Distributed Data Interface	22
FPGA	Field Programmable Gate Array	11
Gbps	Gigabit per second	70
GMT	Greenwich Mean Time	22
GPU	General Processing Unit	11
I/O	Input/Output	36
IP	Internet Protocol	60
IPG	Inter-Packet Gap	59
IPv4	Internet Protocol version 4	5
IPv6	Internet Protocol version 6	5
IRQ	Interrupt Request	20
ISR	Interrupt Sevice Request	23
KBps	KiloBytes per second	80
L2	OSI Layer 2	28
L3	OSI Layer 3	28
L4	OSI Layer 4	28
LKM	Linux Kernel Module	4
LXR	Linux Cross Referencer	38
MAC	Media Access Controller	19

Page

Mbps	Megabits per second	14
N^2d	NIC-to-Disk	2
NAPI	New Application Programming Interface	23
NAS	Network Attached Storage	58
NIC	Network Interface Card	4
NIDS	Network Intrusion Detection System	10
NTP	Network Time Protocol	70
pcap	libpcap packet capture	4
pps	packets per second	67
RAID	Redundant Array of Innexpensive Disks	58
RX	Receive	26
SAN	Storage Area Network	3
SATA	Serial Advanced Technology Attachment	57
SMP	Symmetric Multiprocessing	71
snaplen	snapshot length	32
SNMP	Simple Network Management Protocol	69
SPAN	Switched Port Analyzer	14
SUT	System Under Test	41
TLB	Translation Lookaside Buffer	42
TX	Transmit	26
UDP	User Datagram Protocol	57
VLAN	Virtual Local Area Network	29
VFS	Virtual Filesystem Service	62

PERFORMANCE CHARACTERISTICS OF A
KERNEL-SPACE PACKET CAPTURE MODULE

I. Introduction

THIS chapter introduces the history, state of the art, and current challenges associated with the collection and use of captured network traffic for forensic analysis. It identifies a specific problem and the focus of this research, then it proposes a hypothesis and discusses the design elements of the solution

Section 1.1 highlights the rationale for current methods of network capture. Section 1.2 reviews the problem space and identifies the general direction of this research. Section 1.3 presents the objectives of the research, while Section 1.4 identifies assumptions and limits. Finally, Section 1.6 maps the structure of this document.

1.1 Motivation

Computer networks have vastly improved communication of information, utility of distributed knowledge, and efficacy of the decision process with respect to the timeliness of shared knowledge. The pervasive nature of electronic communications technology has brought with it new levels of interdependence. Rapid distribution of information and resources has provided significant increases in both synergy and efficiency; however, these benefits do come at a cost.

In a conceptual sense, the cost can be described by the adage *A chain is only as strong as its weakest link*. While this philosophical representation of the cost as increased risk is easily understood in the context of network security, the symbolism of a chain falls short in describing the significant and

oft-unknown liabilities of dependence in the massively-connected webs of trust. More concretely, the cost is most readily felt in the strain on resources to establish and maintain the necessary layers of protection required to ensure the intended benefits are experienced by the intended actors.

One such layer of protection is accomplished through the capture and temporary storage of network frames as they traverse boundaries of interest. While many security-related devices capture data, the focus of this thesis is on a NIC-to-Disk (N^2D) capture capability, distinguishing itself by storing the captured data straight to disk. Other than filters used to express traffic of interest, no analysis is accomplished. Packets are captured and written to disk as quickly as possible for analysis, trending, and other forensic *post mortem* activity. While capturing packets alone will not protect the network communications, it serves a vital role in the overall protection of the information and resources that are enabled by the network.

Network capture is cyber hind-sight. It allows network defenders to validate automated alarms—minimizing poor decisions based on false-positives. It provides the ability to reinvent active defenses through *post mortem* analysis of a successful network attack or policy failure. It provides opportunities to improve training through a full-context scenario on once-live data.

Like video surveillance in physical security, the recording process itself does nothing to protect; however, the effect of both the live and forensic analysis provides significant advantages to those who need to validate automated alarms, look for activity that is beyond what the alarm technology is built to detect, and identify the actors, methods, and purposes for a recorded event.

1.2 Problem

The list of benefits attributed to capturing and analyzing network traffic is impressive; but then, so are the resources needed to capitalize on them. As

performance requirements increase, so do the resources required to accurately capture network data and retrieve it. Keeping up with increasing network speed requires defenders to either dedicate greater resources to the task or reduce the amount of resources the task requires.

Commercial solutions tend to incorporate greater resources against the task of getting packets from the network interface to permanent storage. These N^2d capture systems use high speed storage systems, such as a Storage Area Network (SAN), and distribute the capture duties across multiple systems dedicated to the task. This approach is effective but costly. The cost of the hardware, rack-space, power, and cooling to keep it operating properly prevent its use in all but the most critical environments.

Research contributions have led to significant advances in generic network capture—the collection of data from the network to an analysis tool. Efforts have continually reduced bottlenecks and unnecessary capture overhead to get the data to a user-space application, such as Snort or Wireshark. There have been significant gains over the last several years in this direction, but these solutions address only the penalty for transition from the kernel to user-space–leaving untouched the transition from user to kernel-space that is still required for writing the data to disk.

To improve the N^2d efficiency of commodity capture systems, this research will address a relatively untouched portion of network capture architectures–removing the user to kernel-space transition when writing the data to disk.

1.3 Research Objectives

The high-level objective for this research is to improve the performance of an N^2d capture system; however, this thesis will focus on the design, implementation, testing, and analysis of a kernel-mode N^2d capability. The PKAP Linux kernel module will implement a kernel thread which will collect network data

from an in-kernel ring buffer and write it to disk in a series of rotating libpcap packet capture (pcap)-format logs. The capture system, as shown in Figure 2.1 would be placed at a network boundary to capture and record network traffic in support of *post mortem* analysis. The system will capture packets of interest by supporting Berkeley Packet Filter (BPF) filtering, and it will allow for automated user-space retrieval of the packets for analysis.

The primary goals for this research are to construct the first-known implementation of an N^2D capture capability in kernel space, improve the accuracy of network capture by minimizing dropped packets, and reduce the CPU requirements to perform network capture at high speeds.

Building on previous research efforts to enhance the capture side of the equation, the proposed capture system will utilize a modified PF_RING Linux Kernel Module (LKM) [14] to efficiently move the packets from the Network Interface Card (NIC) to a ring-buffer. The PKAP kernel module will retrieve the captured packets from the ring buffer and write them to disk. The PKAP module will implement an N^2D capture functionality wholly inside the kernel with the following objectives:

- Reduce dropped packet rate due to latency between the NIC and the disk
- Reduce Central Processing Unit (CPU) utilization of the capture process
- Reduce memory utilization of the capture process
- Retain user-space retrieval performance of the stored packets

1.4 Limitations

The following is a list of limitations for this research:

- This thesis will not address any legal issues with capturing this data. It is assumed that the legal authority is in place to perform both collection and analysis.

- This effort will target only the capture of data and its retrieval. It will not provide research into how to analyze the data beyond the rationale for the capture system accessing methods and performance requirements.
- The PKAP proof of concept will implement only Internet Protocol version 4 (IPv4)-based structures. This will allow prototyping with known protocols and existing network data against existing tools. Ideally, an IPv6-based structures would be used, given that IPv4 packets can be stored within an Internet Protocol version 6 (IPv6) addressing scheme. . . however, this would complicate the process greatly for a proof that is not yet known to provide the requisite performance gains to pay for the complexity. Additionally, the existing PF_RING work is not yet capable of IPv6 capture and would require significant efforts to include the functionality. The use of IPv4 during this effort will not directly alter the findings for the N^2d capture; however, it will have some effect in the workload necessary to create the `sk_buff` kernel structure and to execute packet filtering. Since filtering is a required capability, the workload should affect any capture method similarly.

1.5 Implications

A PKAP-based capture system will reduce unnecessary overhead in writing captured packets to disk. The improved efficiency will allow higher network rates to be captured through improved efficiency with existing hardware. Reduced cost would potentially be seen through some combination of the following: reduced capture system hardware components, reduced rack-space, reduction of power and cooling requirements, and improved accuracy of analysis. Improved efficiency at this low level will also induce improved top-end performance of distributed systems, allowing capture capabilities to more closely follow the speed limits of storage. Considering the vast investment by Law Enforcement, Homeland Defense, the Department of Defense, and industries

such as financial and medical institutions, any reduction of unnecessary overhead will improve the cost to benefit ratio.

1.6 Thesis Structure

Chapter II provides background information about packet capture and presents the state of relevant research on the subject. Chapter III delves into the details of designing the PKAP kernel-space N^2d capture capability, and Chapter IV provides the methodology used to conduct the performance analysis of the capture system. Chapter V presents the results and analysis of the system performance, and Chapter VI identifies the conclusions drawn from the performance analysis and indicates future issues in this research area.

II. Literature Review

THIS chapter presents background information and applicable research related to network forensics, capturing packets, Linux kernel internals, and storage. Due to both legal and financial constraints, the research area concerning the behavior and performance of N^{2}d capture has received little attention in any direct form. That said, there is a significant amount of historical and ongoing research in the related areas of generic packet capture, storage performance, filesystem design, and operating system efficiency. Section 2.1 provides a brief introduction to capturing packets—including common terms, types, and tools. Section 2.2 relates the general packet capture concepts to network security, while Section 2.3 provides a deeper discussion on the mechanics of current packet capture implementations and advances.

2.1 Background On Packet Capture

2.1.1 Sniffing. Sniffing is a term commonly used to identify the promiscuous retrieval of network traffic. To sniff packets from a network, a physical device must provide a copy of the data traversing the network, and the sniffer must be able to capture the data without the network hardware, firmware, drivers, network stack, or sniffing application dropping a packet that is not addressed to the sniffer system itself. Sniffing and capturing are sometimes used synonymously; however, sniffing typically indicates analysis, while capture can indicate both analysis and storage.

The common method for sniffing the network is through the use of the libpcap library. According to the CHANGES file distributed with libpcap, version 0.0 of libpcap was released on 20 Jun 94. As of the writing of this thesis, the latest official release of the mainstream libpcap library was in 2008 [16], though there are also several prominent forked libraries as well. Section 2.3 will discuss the mechanics of libpcap in greater detail.

2.1.2 Capture Types. There are a wide variety of reasons to capture network packets. While some uses require only the header information of a packet, others require the full packet. Capture of full packet is significantly more difficult than capturing only headers; yet, it continues to enable applications that require only the header. This research proposes to increase efficient handling of the capture data in order to prevent loss of data. To continue pursuit of this end, the remainder of the document discusses only full-packet capture.

2.1.2.1 Live Analysis. Live analysis is the most significant use of captured network data. To be analyzed live simply means that the packets fulfill their purpose as they are retrieved. That purpose could be simply ensuring that network packets have gotten to a specific point, or it could be a detailed, context-sensitive analysis of a network session to detect network intrusions. As seen in the evolution of packet capture technology presented in Section 2.3, this category of capture end-use is the principle driving force behind most current research—getting captured packets from the disk to the analyzer process as quickly as possible.

2.1.2.2 Cyber Hind-Sight. There is not a generally accepted term to address this packet capture end-use. Cyber hind-sight is a description of the rationale for employing this type of capture, but this thesis refers to this activity as N^2d capture. The only parsing or analysis of the packet that is accomplished upon retrieval is simple filtering to discard any packets that are not of interest. The capture mechanism exists not to analyze, but to write the packet data out to disk for later use.

As mentioned briefly in Chapter I, the N^2d capture capability provides a resource similar to that of video cameras. It allows incident handlers to evaluate methods, identify further detail after the fact, and even glean motives based

upon expert analysis of the recorded behavior. Not only does the capture assist in determining what has happened, it also provides the hind-sight necessary to alter both architecture and policy to prevent it from happening again.

The performance characteristics that impact live analysis directly impact N^2D capture as well. It is from the point that the packets are available to the capture application that the computational challenges differ significantly between live analysis and cyber hind-sight. While the post-capture challenges for live analysis tend to be memory size constraints and CPU speed, the challenges for cyber hind-sight tend to be bus speeds, I/O overhead, and shared access to a limited storage device.

2.1.3 Capture Tools. The software application that orchestrates the capture process and ultimately what to do with the captured packet is the capture tool. Most common tools link to libpcap for the actual capture functionality, but there are some that do not.

2.1.3.1 Common Tools. The few mentioned here are representative of the category, but in reality, there are far too many capture tools to list. The capture tools in this category all use the libpcap library to accomplish the capture process, and all the ones listed are open-source projects that are available to anyone. In fact, some are packaged by default with common Linux distributions.

TCPDUMP is a command line utility that has been commonly distributed with Unix-based systems since version 2.0 was made publicly available in 2001. It has been the standard packet sniffing tool for system administrators, network administrators, network security personnel, academic researchers, hackers, and crackers. It is capable of both live analysis and N^2D capture, but it is diminished as the best options for either use today. TCPDUMP is still a useful

staple of Unix-based distributions, but most users wanting a generic protocol analyzer have turned to more graphical or sophisticated tools like Wireshark.

Wireshark began life in 1997 under the name Ethereal, when Gerald Combs started its development in an effort to better understand networking; it was renamed to Wireshark in 2006 [22]. The authors list for Wireshark currently contains over 600 contributing authors, and the number of contributions grows as features are desired or protocol dissectors are needed to support research. Wireshark provides the default graphical interface, a command line version of the full utility named tshark, and a host of small pcap utilities. Wireshark is also capable of both live analysis and N2d capture; however, it is not an efficient option for the latter.

Snort is one of the best known Network Intrusion Detection System (NIDS) available today; yet it also began life in 1998 as a simple sniffer. Martin Roesch first created Snort to gain experience with cross-platform libpcap development and to more easily see the application layer of captured traffic. Over time, Snort developed into one of the most capable full-blown NIDS available today [31]. Snort provides highly customizable analysis and alarming for live packet capture, and it also provides the ability to cobble together an easily batch-driven N2d capability—though somewhat cumbersome in dependencies, configuration complexity, and program size for this purpose.

DaemonLogger was specifically designed to be a lightweight N2d capability. Also written by Martin Roesch, DaemonLogger is a libpcap-based capture service with an extremely straightforward and focused code base for the purpose of both writing captured packets to disk and managing the rotation of capture files for long-term capture tasks [32]. Though many tools exist that could be used for the long-term capture of packets to disk, this is one of the few applications that was designed to do that alone—and do it with the least amount of complexity and overhead possible. The efficiency of code and func-

tion made it the clear choice to represent user-space N_d^2 methodologies for this thesis's performance analysis.

2.1.3.2 High-End Capture Tools. An abundance of more highly specialized commercial capture tools also exist today. These tools may or may not use libpcap; some customize the mainstream library, while others forsake it completely. These high-end tools often employ purpose-built hardware to provide extremely high network capture rates, albeit at the cost of a much higher investment.

Products in this category can provide significant advances in the efficiency of collecting packets of interest from the network to the analysis application. The typical approach is to use hardware-based network classification such as Network Processors, Field Programmable Gate Arrays (FPGAs), Application Specific Integrated Circuits (ASICs), or multiple cores in a General Processing Unit (GPU) to identify a filtered data stream of interesting traffic at line rates before the packets even touch the capture application host's system bus or memory. Also commonly provided is a custom version of libpcap that provides Direct Memory Access (DMA) from the specialized hardware to the capture application. Some tools provide transparent load balancing of the captured packets to support data rates higher than a single host system could consume, while a few commercial tools provide acceleration of the analysis tool itself—whether that tool be Snort or some other libpcap-based tool.

These steps all provide highly optimized capture paths to sniff packets from the network and provide them to a live analysis tool; however, the capture optimizations employed can actually serve to make the problem with storage more apparent. Storage technology is not able to keep up with data transmission or computation rates. Optimizing the capture portion of an N_d^2 capability only further exacerbates the situation where storage is not able to consume the network traffic. At the moment, the solution is to distribute the packets to be

captured to a sufficient number of capture devices to meet the network load. This solution chooses to throw more resources at the problem, and it will work for those with enough resources to purchase and utilize the SANs and server farms required.

2.2 NIC-to-Disk Packet Capture and Network Security

2.2.1 Purpose. Network security is an umbrella term that covers many tools and policies, with the ultimate goal of ensuring that networked resources are available only to those people or systems that should have access to them when they should have access. Information Security is the essence of network security, and it is commonly described by a triad of concepts: confidentiality, integrity, and availability [7]. While this triad has been extended by some to include more specific, and somewhat overlapping concepts (i.e., Authentication, Non-Repudiation, etc.), the triad summarizes the common goal well and succinctly.

- Confidentiality: concealing information or resources so that only those authorized have the ability to access them
- Integrity: ensuring that the information or resources have not been altered without being detected
- Availability: making the information or resources available to those who need them when they need them

Full-packet network capture is one network security tool. Capturing packets from the network to a storage device is a powerful enabler. According to the computer crime experts at Foundstone, Inc., network-based evidence allows an organization to accomplish a number of tasks [25]:

> - Confirm or dispel suspicions surrounding an alleged computer security incident
> - Accumulate additional evidence and information

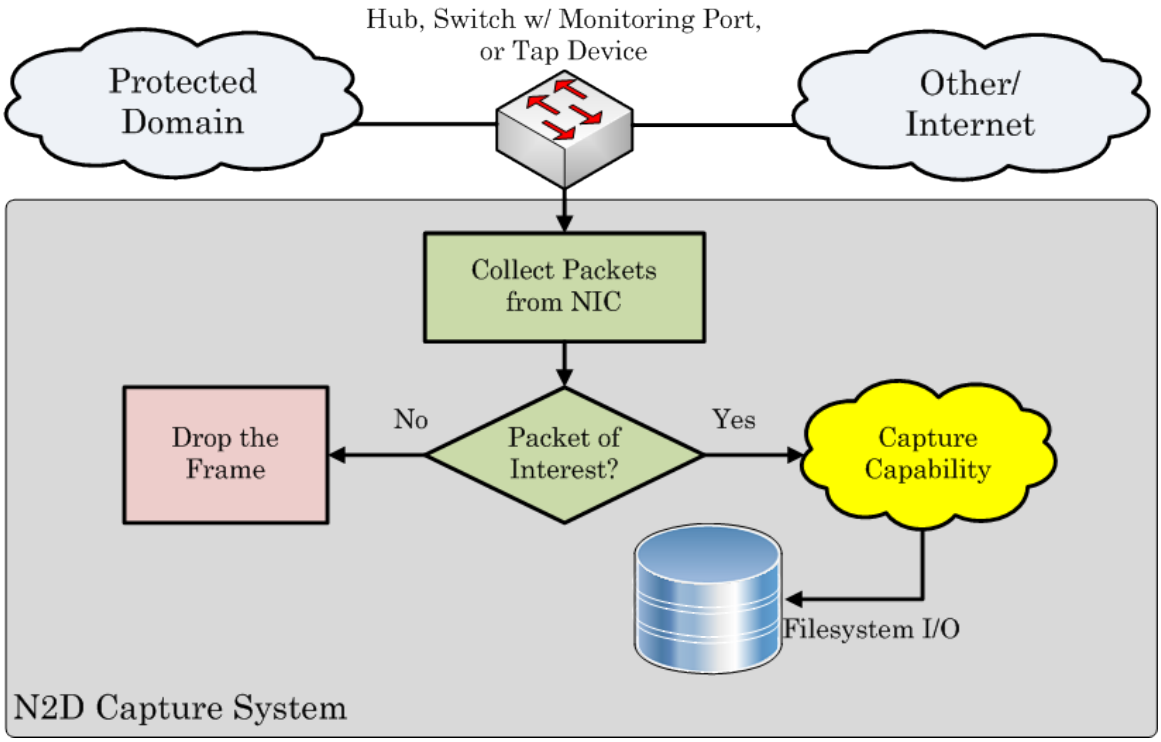


Figure 2.1: Simple NIC-to-Disk Capture System Architecture

- Verify the scope of a compromise
- Identify additional parties involved
- Determine a time line of events occurring on the network
- Ensure compliance with a desired activity

2.2.2 Execution.

2.2.2.1 Basic Architecture. Figure 2.1 illustrates the basic architecture of an N2D capture system. One of the simplest implementations of an N2D capture capability is to install Linux on a spare computer system, connect that system to a hub between an internal network of interest and its default gateway, and run software such as TCPDUMP to identify packets of interest and store them to disk in a pcap-format file. This example is better suited to troubleshooting a very small and slow network than providing a network security capability; however, the core pieces are present: 1) A place to plug in that provides a copy of all the packets transmitted across a boundary of interest (the

hub), 2) A networked system that can be placed in promiscuous mode to receive packets not destined for itself, 3) a filtering capability to sniff only those packets that meet identified criteria for capture, and 4) software that will write the captured packets to disk according to a standard format that will allow other tools to parse and analyze the packets later.

The pcap-formatted file mentioned above is referring to the generally accepted standard for network capture. This standard is the format of the libpcap capture library that is used almost universally to obtain raw packets from a system's NIC. The library is available on all common operating systems, to include Microsoft Windows in the form of the WinPCAP driver. It presents a standard Application Programming Interface (API) for the preparation and use of a network device to promiscuously sniff traffic as well as to store the traffic to disk. Section 2.3 will discuss libpcap in much greater depth, but the important thing for now is that an N^2d system must use a standardized file format to take advantage of existing tools suites.

In times past, Ethernet or Fast Ethernet hubs were the tools of choice to sniff packets; however, the current network standard is Gigabit Ethernet, and hubs do not exist for that environment. Using a hub will impose a harsh penalty on the network link it is used on, forcing a limitation of 100 Megabits per second (Mbps), so for most environments the only two options to obtain a copy of the packets is through the monitoring port (sometimes called the Switched Port Analyzer (SPAN) port) of a managed switch or through a specialized network tap device. Determining which of the two is most appropriate depends on the following factors: network speed (Gigabit, 10-Gigabit), physical network type (fiber, copper), and number of capture interfaces required. Other options exist for wireless networks, load-balanced sniffing, etc., but they are not the common case. In medium to large networks, using switch monitoring ports is ideal, since they are 1) already part of the infrastructure and 2)

dynamically configurable to provide both local and remote sniffing throughout the infrastructure.

It is also possible to place the capture device inline, either as a routing device or transparent bridge. This option is seldom chosen since the capture system becomes a single point of failure with reliability significantly less than dedicated network infrastructure. High end systems with specialized network hardware and purpose-built processing can provide inline capture and storage functionality with the reliability of dedicated network infrastructure; however, this approach is beyond the scope of this research.

2.2.2.2 Extended Architecture. While Section 2.2.2.1 depicts the principle workload of the capture system, it is important to understand that the capture of packets to disk is only an internal service. The external, and most significant, service provided by an N^{2}d system is the actual retrieval and utilization of the captured packets. Figure 2.2 depicts the extended architecture of the system. Though the extended architecture is not directly addressed by this research, its purpose for and methods of extracting data from the N^{2}d collector is relevant in the endeavor to streamline the collector system.

Once the data is captured and stored on the collector, it remains there until requested or expired. The request could be automated, triggered by an alarm; or it could be manual, generated by an analyst attempting to dig further into a possible incident. Other requests could come from archival systems or data-mining processes. Regardless of the source, the collector must be capable of retrieving the requested data and sending it to the requester without compromising the ability to continue reliable packet capture. The retrieval process can vary; however, the action typically involves processing the capture files through filters to build custom capture files containing only the connection(s) requested. Parsing the captured files introduces a read workload to the sys-

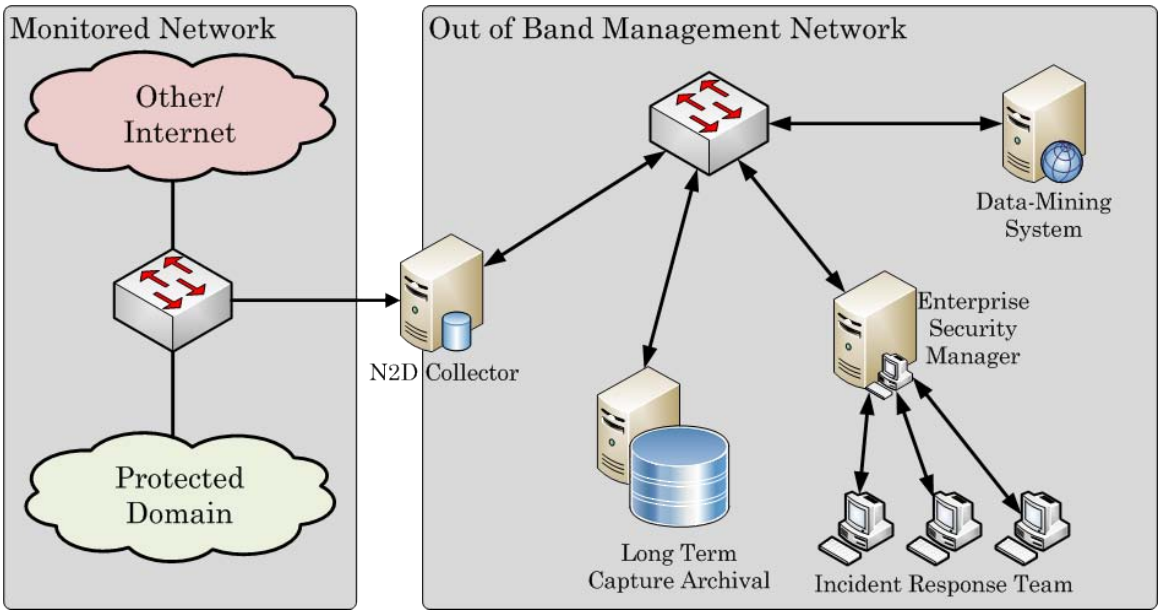


Figure 2.2: The Extended Architecture of a NIC-to-Disk Capture System

tem that could interrupt the task of writing captured packets to disk or become blocked, hindering the satisfaction of the request for captured data.

2.2.3 Cost. In reality, security is an acceptable probability of risk, as determined by the balance between the cost of a threat exploiting a vulnerability and the cost of preventing the exploitation. There are many combinations of tools and policies to achieve a level of acceptable risk; in the case of network security, the tools may extend from deploying anti-spam protection to employing an entire Computer Emergency Response Team (CERT), capable of full-time network defense. The prudent quantity of resources invested in protection, detection, and response technology is determined by the risk—the combination of the serious nature of the threats and the impacts of the vulnerabilities, should they be exploited.

The cost of the N^{2}D collector itself can be significant for large networks, but it is eclipsed by the investment in the extended architecture and personnel required to make use of it. For this reason, N^{2}D capture systems are typically

employed only when the threats are well-resourced and the cost of exploitation is very high. The total costs include:

- Network infrastructure that supports dedicated capture: procuring the hardware that allows sniffing (hubs, SPAN-enabled switches, dedicated taps, etc.) and building networks that meet sniffing requirements without destroying desirable redundancy
- Capture systems capable of ingesting the network data rates
- Storage devices/systems capable of simultaneous write and read speeds required of the system
- Related systems necessary to take advantage of the captured date (data mining, alarming, etc.)
- Personnel trained to properly use the recorded traffic—both technically and legally
- Physical infrastructure to support all the above (power, cooling, space, etc.)

2.2.4 Technical Challenges. The performance characteristics of the capture process impact both the cost and the scalability of the system. The principle performance challenge of an N^2d system is that the cost for the storage system adequate for a given bitrate is far greater than the cost for the network infrastructure required to sustain that same bitrate.

Illustrating the problem with capturing network data in real-time, a team of researchers trying to increase the performance of a Sprint network accounting tool stated, “We already have in place an OC-48 passive monitoring system for capturing and storing a detailed record for every packet. But because of constraints on storage and bus bandwidth this will not be feasible at 10 Gb/s and above.” [20] The main purpose of the network accounting tool is to capture network traces, which are representative packet flow records for every packet

stream traversing a network. The researchers perceived the practical limit of the current capture mechanism to be roughly OC-48, after which they would need to adapt the format and compression of the data to get past bus limitations. The team estimated that "Disk array speed cannot keep up with link bandwidth. At OC-192 speed, a packet-level trace would require a disk bandwidth of roughly 250 MB/s (assuming 300 Byte packets and a 64 Byte long packet records)." [20]

N^2d capture devices must store more than just the flow records; they need to capture full packets. An N^2d system could not rely on a 5:1 reduction from transmitted data to stored data. In fact, an N^2d system must store more data than is transmitted on the wire to account for the 16 byte per packet header for the pcap standard. Compression could be used to store redundant packet header data per session; however, the pre and post processing required to store and retrieve the data would likely increase the latency between capture and storage rates.

2.3 Packet Capture Mechanics

This section discusses the mechanics of current packet capture techniques and the direction of recent advances to the technology. The fundamental basis for most techniques is rooted in libpcap, so the following sub-sections will begin with the mainstream libpcap and discuss significant changes that have improved overall capture performance.

2.3.1 Overview. The libpcap library has existed as a public, Berkeley Software Distribution (BSD)-licensed library since version 0.0 was released in 1994, and it is included as a standard library with many Unix and Unix-like operating systems [16]. Though the library can be installed on most operating systems in use today, the background presented here will focus on the Linux implementation.

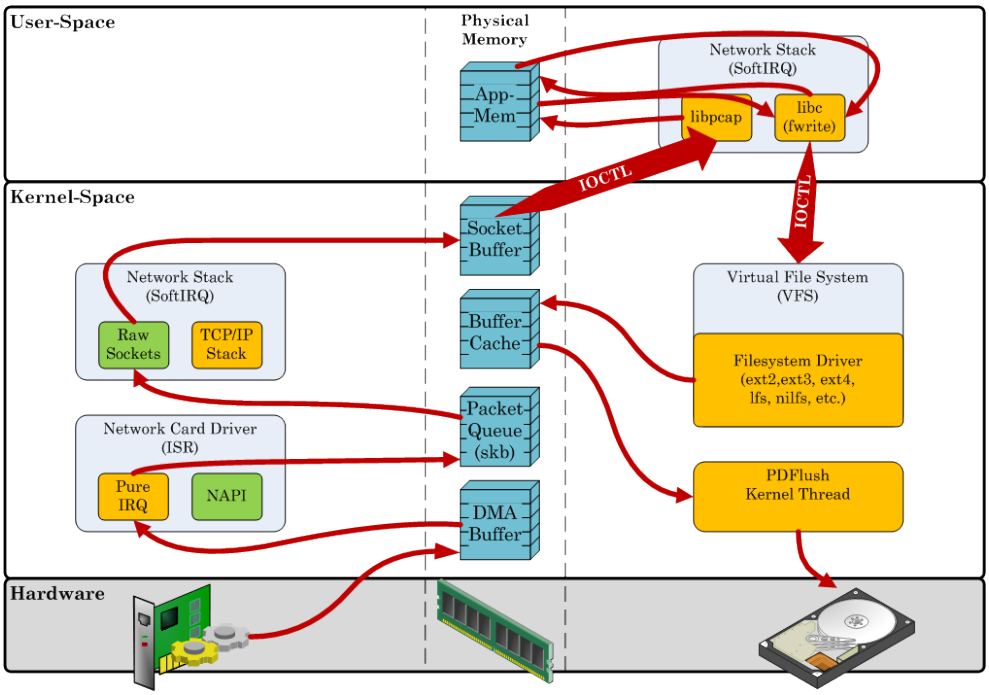


Figure 2.3: The libpcap-1.0.0 (mainstream) Capture Process: Significant use of memory and queuing provide low performance and difficult measurement of dropped packets

2.3.2 AF_PACKET (libpcap-1.0.0 vanilla).

2.3.2.1 Packet Path. Figure 2.3 depicts the path that packets take when a capture application is linked with the mainstream libpcap library. There are many additional paths and functions that are associated with the reception of a packet from a network interface; however, this discussion is limited to the paths relevant to promiscuous capture using the standard libpcap methods on Linux.

The packet is copied to various segments of system memory no less than five times before it can be written to a storage device. Packets captured and stored via the mainstream libpcap will follow this path unless they are filtered out. Packets can be filtered in one of two common ways: 1) the NIC can filter out packets not destined for its Media Access Controller (MAC) address if not in promiscuous mode, and 2) a BPF could be used to discard packets that match

specific patterns. The following describes the steps of the packet from the NIC to the disk:

1. The NIC uses a DMA transfer to place the frame into the DMA buffer assigned to it and triggers a hardware Interrupt Request (IRQ) to let the driver know that a frame has been placed in the NIC's assigned DMA buffer space [4].
2. The NIC driver has registered a callback function, called an interrupt handler, for the specific IRQ assigned to the NIC. This handler retrieves the frame from the DMA buffer, inserts the raw frame data into a socket buffer (`sk_buff`), initializes some of the `sk_buff` parameters and schedules a `NET_RX_SOFTIRQ` soft-IRQ for the network stack to pick up the `sk_buff` from another buffer space in kernel memory [4].
3. The network stack uses a function named `netif_receive_skb()` to handle the `sk_buff` placed in the queue by the NIC driver, and this function will pass a copy of the `sk_buff` to the receive queue of any protocol taps—these initiated by libpcap's creation of an `AF_SOCKET` socket, allowing the `sk_buff` to bypass layer 3 and layer 4 protocol handlers [4].
4. Packets in an `AF_SOCKET` protocol handler's receive queue are retrieved one at a time via libpcap's `pcap_read_linux()` call and passed to the capture application callback functional [16, 38].
    - This process uses a `recvmsg()` syscall on the socket descriptor obtained when libpcap created an `AF_SOCKET` socket.
    - The result is an expensive kernel-space to user-space copy.
5. For an N^{2}d application, the callback function will usually gather relevant traffic statistics, perform pcap log file rotation as needed, and call the libpcap `pcap_dump()` function which uses the `fwrite()` function from the glibc library to begin the transfer of the packet to a storage device [16].

- `fwrite()` buffers the data to be written to disk, so the packets are copied into a user-space buffer until the buffer is full [27].
- The buffering requires additional copies, but the user-space to user-space copies significantly reduce the number of syscalls and associated user-space to kernel-space copies—ultimately improving write performance beyond what is capable without buffering [19].

6. When the write buffer is filled during an `fwrite()` call (or `fsync()` is called), the buffer is written in a single `write()` syscall and copied from user-space to kernel-space inside the Linux kernel's `vfs_write()` call [38].
7. The VFS transparently calls the write file operation provided by the driver for the filesystem containing the pcap dump file, and the driver writes the buffer data to the buffer cache and marks the page dirty [26].
8. The Linux kernel utilizes `pdflush` worker threads to write dirty pages from the buffer cache to the storage device [26].

Excepting a poorly written capture application, this describes the worst case path through the Linux implementation of the libpcap library.

2.3.2.2 The pcap *File Format.* Due to the longevity of libpcap and the fact that the library is the standard for packet capture, the format that libpcap uses to save packet data to disk has become the *de facto* standard for network capture data. The basic pcap file format has remained unchanged since 1998 [12]. There are alternatives that provide better clock granularity or other benefits; however, the pcap-format is generally used in order to take advantage of the myriad of libpcap-based third-party applications that exist for collection, analysis, and manipulation.

The format specifies both a global file header and a per-packet header to identify contextual information that would otherwise be lost if only the frame

itself was written to disk verbatim. Figure 2.4 [5,12,16] graphically depicts the layout of the headers within a pcap-formatted file and provides both the global and per-packet header structures.

The global header is 24 bytes long and includes seven fields to identify the characteristics of the capture file. The `magic_number` field is used to detect the byte ordering of the file. All systems, regardless of byte ordering, write the same number to the first data field of the pcap file. When the file is read, the reading application can determine the byte ordering of the file by the on-disk representation of the magic number and treat the remainder of the file accordingly. The global header also includes the major and minor version numbers to ensure the same file format is used to both create and read the capture file. The `thiszone` field contains signed representation of the time shift from Greenwich Mean Time (GMT) in seconds; however, this is typically set to 0 in practice. The `sigfigs` field is unused and always set to 0, but its intended use was the identification of the accuracy of the time stamps. The `snaplen` field reveals the snapshot length of the libpcap library when creating the file; the snapshot length is the maximum number of bytes captured. The smaller of the frame size or the snaplen is the maximum number of bytes of the frame saved to the disk. The `network` field identifies the type of network frames the file contains; this could be one of numerous types (i.e., Ethernet, Token Ring, Fiber Distributed Data Interface (FDDI), etc.); however, the field is most commonly set to 1, which represents Ethernet. [12]

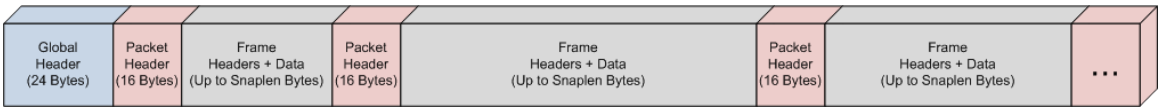
The per-packet header is 16 bytes long and includes three fields—one of which is a `struct timeval`, containing two 32-bit fields. The `timeval` structure contains the fields `tv_sec` and `tv_usec`, which represent the time the packet was captured to within microsecond granularity. The `tv_sec` field is the number of seconds since January 1, 1970 00:00:00 GMT, and the `tv_usec` field represents the number of microseconds from the current `tv_sec`. This granularity of time is the most limiting factor of the current libpcap file format,

and it is the most commonly adapted portion of the headers in alternative capture formats. The `caplen` field represents the number of bytes of the packet that was written to the file, which allows the file parser to jump to the next packet header. The `len` field represents the original length of the packet. If the `caplen` is sufficiently long, the `len` and `caplen` fields will be equal; however if the `caplen` is shorter than the `len`, then the reading application knows that the packet was truncated by the capturing application's snaplen setting. [12]

2.3.3 NAPI. Figure 2.5 depicts the path that packets take when a capture application is linked with the mainstream libpcap library and the Linux kernel is configured to use the New Application Programming Interface (NAPI) enhancement to network device scheduling. NAPI affects the capture process at a much lower layer than libpcap, and its impact is more broad than packet capture alone. NAPI significantly reduces CPU overhead through a hybrid of polling and signaling for network card drivers.

Prior to NAPI, network drivers had two mutually exclusive options to retrieve packets from the DMA buffer assigned to the NIC. The first option required the NIC to trigger an Interrupt Sevice Request (ISR) (hardware/real IRQ), for which the device driver would have registered a callback function. Using this method, every packet received would require an IRQ; this is very cheap for network devices with very low utilization, but it creates an IRQ storm during high usage. This storm would debilitate even the most well-resourced systems. The second option utilized device drivers that would poll the DMA buffer periodically, rather than registering a callback for the NIC's ISR. While the polling performed very well under high loads, it tended to waste considerable CPU cycles during periods of low network use. [4]

NAPI is not itself a new way to retrieve packets; rather, it is the combination of the best qualities of signaling and polling. Under periods of high use, the driver automatically shifts to polling. When the polling activity re-




```
/*
 * Global Header Structures: As used in pkap.h after adaptation
 *                           from the libpcap-1.0.0 library
 */

// from savefile.c
#define TCPDUMP_MAGIC   0xa1b2c3d4

// from pcap.h
#define PCAP_VERSION_MAJOR  2
#define PCAP_VERSION_MINOR  4

struct pcap_file_header {
    uint32_t    magic;
    u_short     version_major;
    u_short     version_minor;
    int32_t  thiszone;      /* gmt to local correction */
    uint32_t    sigfigs;        /* accuracy of timestamps */
    uint32_t    snaplen;        /* max length saved of each pkt */
    uint32_t    linktype;       /* data link type (LINKTYPE_*) */
};
```

```
/*
 * Packet Header Structures:  As used in pkap.h after adaptation
 *                            from the libpcap-1.0.0 library
 */

// from pcap-int.h
// NOTE: This definition is important, since the system timeval
// uses 64-bit integers...
struct pcap_timeval {
    int32_t     tv_sec;     /* seconds */
    int32_t     tv_usec;    /* microseconds */
};

struct pcap_sf_pkthdr {
    struct pcap_timeval ts;     /* time stamp */
    uint32_t            caplen; /* length of portion present */
    uint32_t            len;    /* orig length of packet */
};
```

Figure 2.4: The Libpcap File Format: Depicting the global and per-packet header placement and structure

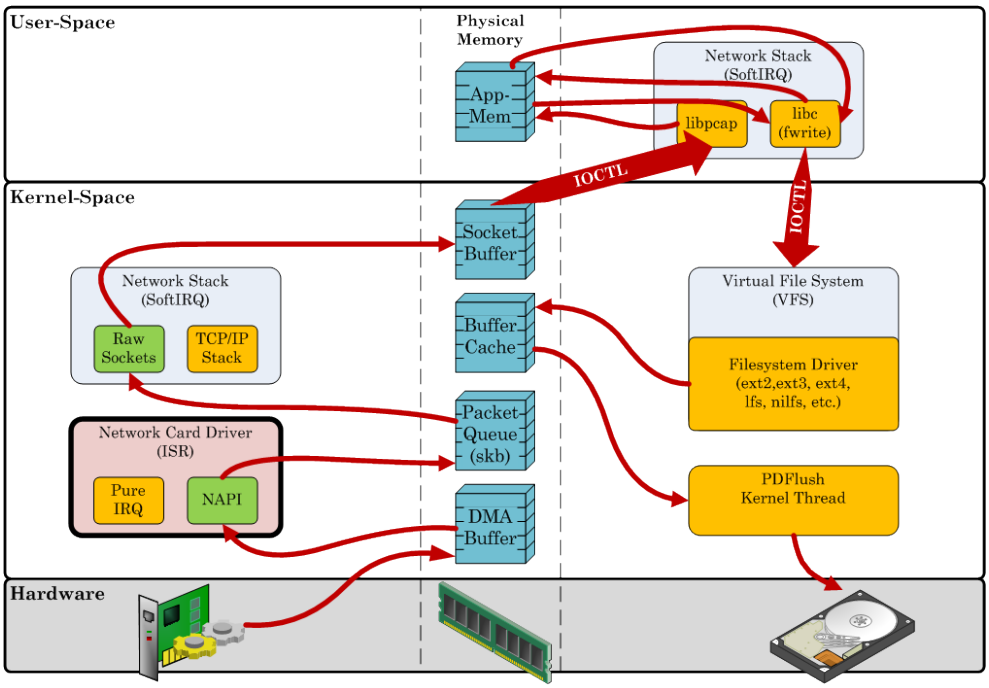


Figure 2.5: The lipcap-1.0.0 + NAPI Capture Process: The memory usage has not changed, but the changes to the NIC Driver to utilize a hybrid Poll/IRQ process saves significant CPU time

turns no data too many times in a row, the driver will switch to a signaling method by registering the callback for the ISR. Then once a new packet arrives, the handler will unregister the ISR handler and begin polling again. It is the demand switching between signaling and polling that allows a driver to use the minimum CPU resources to receive packets for whatever the network load circumstances may be. [4]

2.3.4 libpcap-mmap. Figure 2.6 depicts the path of packets through a capture system using a memory mapped ring buffer to remove the kernel-space to user-space copy penalty. The memory mapped ring optimizations are now available in the version 1.0.0 mainstream libpcap library. For libpcap to utilize the memory mapped ring buffer, the AF_PACKET protocol handler of the Linux kernel must be modified to build and share the ring with the user-space library. These modifications have become part of the official Linux kernel sources, but

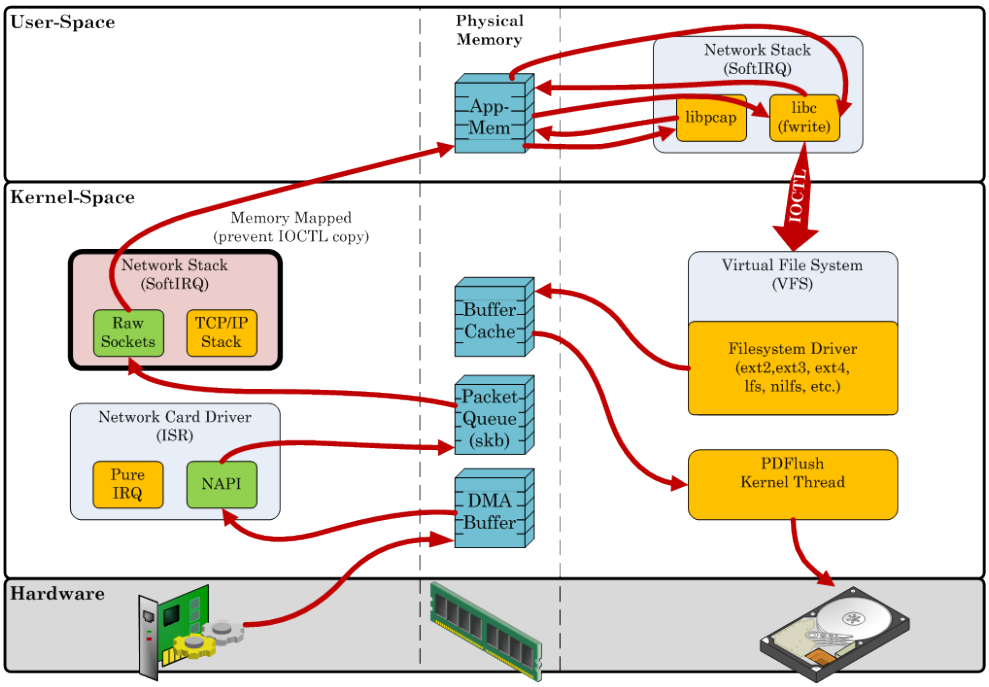


Figure 2.6: The libpcap-mmap + NAPI Capture Process: Removed the expensive kernel-space to user-space copy by memory mapping the kernel-space memory into libpcap's user-space

the configuration option may or may not be compiled into the default kernel binaries of a particular Linux distribution.

The memory mapped approach is similar in concept with, but implemented differently than, the PF_RING approach discussed in Section 2.3.5. Libpcap-mmap appears to optimize for a more general use of raw sockets; whereas, PF_RING seems to be optimized more specifically for passive capture. The libpcap-mmap library and associated modifications to the AF_PACKET protocol handler provide both Receive (RX) and Transmit (TX) memory mapping; however, all packets still flow from the NIC drivers receive function to each of the protocol handlers. This approach differs from PF_RING; additional discussion will follow in the detailed description of the PF_RING approach.

2.3.5 PF_RING: libpcap-ring. Figure 2.7 depicts the path of packets through a capture system using the PF_RING memory mapped ring buffer.

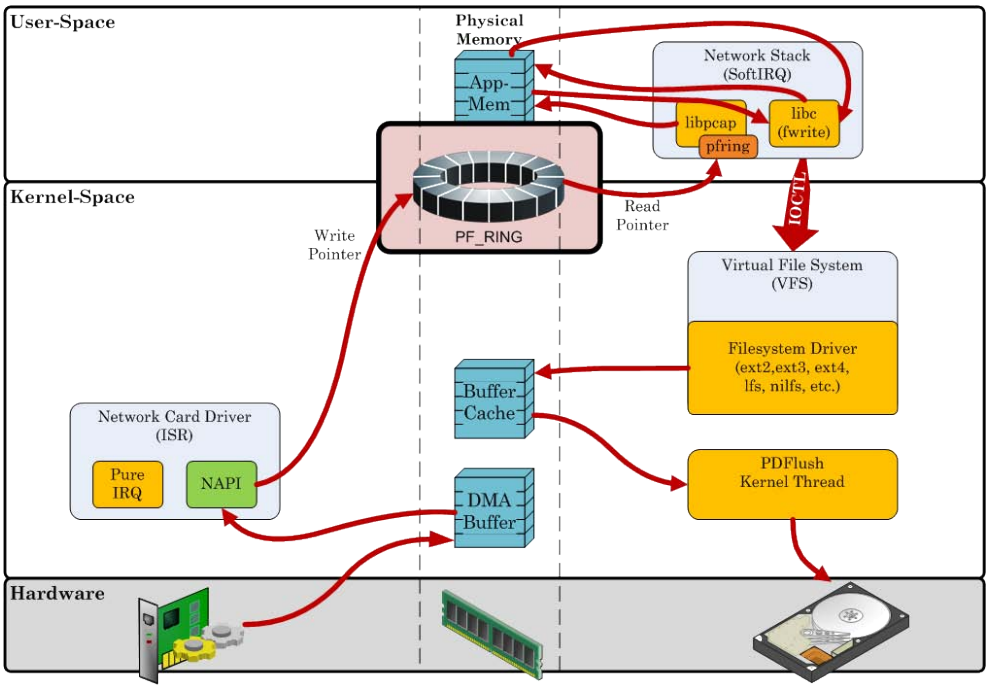


Figure 2.7: The libpcap-ring + NAPI Capture Process: Establishing a PF_RING protocol prevents the packets from going through the network stack, allowing the NIC driver to deliver the skb directly to a ring-buffer that is accessible by the user-space libpcap-ring library

PF_RING is currently at version 4.1.0, and it includes a modified version of the libpcap-1.0.0 library [14]. The modified libpcap (libpcap-ring) allows applications that utilize libpcap to take advantage of the PF_RING advances without requiring changes to the application's source code.

Though using the libpcap-ring library is advantageous for porting existing libpcap-based applications, using PF_RING through libpcap is not required, as all the interaction with the PF_RING kernel module occur through the pfring user-space library. The pfring library is included in the PF_RING source tree, and it can be used by a user-space application directly. In fact, libpcap-ring does not directly manipulate the ring buffer; the modifications to libpcap-ring provide access to the ring buffer through the pfring library. Use of libpcap-ring requires no source code changes on the part of a libpcap application; however, the application must be recompiled with the libpcap-ring library to accommodate correct linking [15].

2.3.5.1 The PF_RING Kernel Module. The PF_RING kernel module is a dynamically loadable Linux kernel module that implements the PF_RING protocol handler. The PF_RING protocol handler resides at OSI Layer 2 (L2), the same level as the AF_PACKET protocol handler. Rather than force modifications to the AF_PACKET protocol handler in the official Linux kernel source tree, PF_RING created its own handler, so that changes could be made without concern that they would break other users of the protocol [15].

The PF_RING protocol handler can receive packets in the same way that AF_PACKET receives packets—inserting itself into the list of waiting packet sniffers that will get copies of the incoming packet. This method allows protocol handler to get a copy of the packet without forcing the packet to be processed through the OSI Layer 3 (L3) or OSI Layer 4 (L4) network stack functions. Additionally, if used with PF_RING-aware network device drivers, the PF_RING

kernel module can receive packets with a direct hook—bypassing the standard `netif_receive_skb()` function and the additional workload it would entail.

The PF_RING module registers its protocol handler using the `dev_add_pack()` function. This action effectively places the PF_RING `packet_rcv()` function in the list of handlers that the `netif_receive_skb()` copies packets to as the NIC driver receives packets from its device. This method is necessary if other non-PF_RING-based capture tools are being used, the system requires standard network communication outside passive packet capture, or if bridging or Virtual Local Area Network (VLAN) activities are required.

For cases where the above mentioned activities are not required and a PF_RING-enabled NIC driver can be used, PF_RING can use its registered hook to bypass the `netif_receive_skb()` duties and send the `sk_buff` directly to the PF_RING handler. Figure 2.8 steps through the conditions used by the PF_RING-enabled Intel e1000e NIC drivers—showing that the hook->ring_handler() can ingest the packet and cause the NIC's receive function to return prior to any standard Linux networking calls, specifically `netif_receive_skb()` [14].

Bypassing the core Linux packet reception function, `netif_receive_skb()`, prevents general use of the NIC when PF_RING is enabled, but the benefit is improved efficiency of passive capture by forgoing the processing time required to perform the principle duties of the `netif_receive_skb()` function call, which are [4]:

> - Passing a copy of the frame to each protocol tap, if any are running
> - Passing a copy of the frame to the L3 protocol handler associated with `skb->protocol`
> - Taking care of those features that need to be handled at this layer, notably bridging

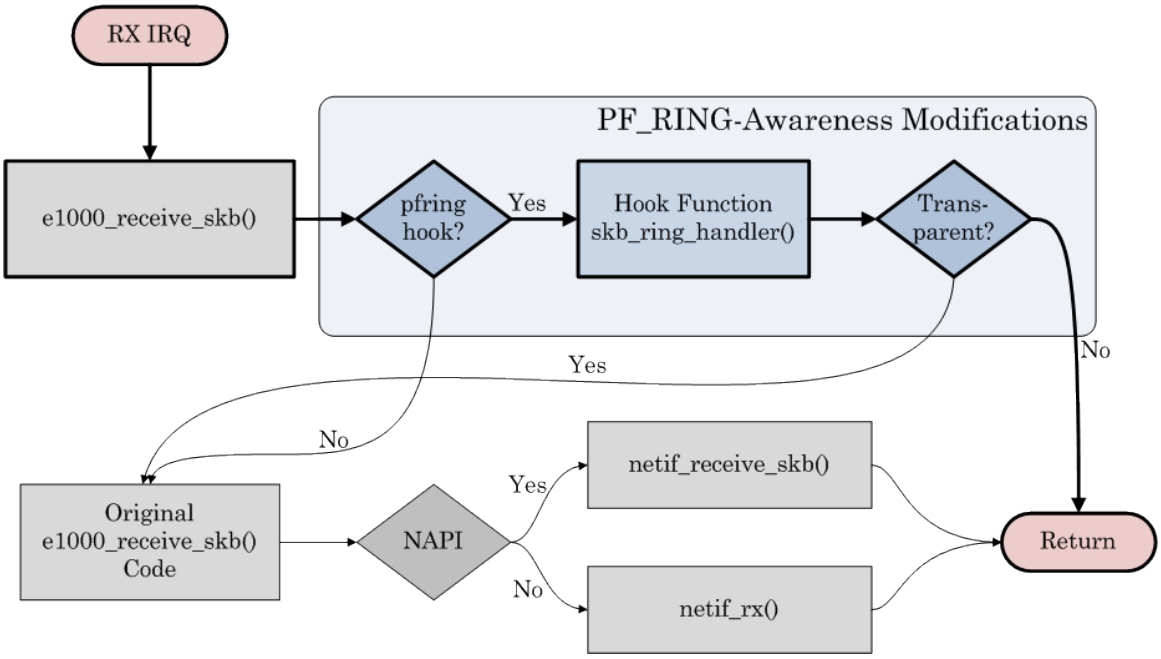


Figure 2.8: PF_RING-Aware NIC Driver Flow: The bold path is the optimized packet path through the PF_RING-aware NIC driver for passive sniffing

Once the PF_RING module has the packet, whether by protocol handler or protocol hook, it provides the following processing options, as depicted in Figure 2.9:

1. Optionally captures TX packets
2. Parses out the necessary portions of the `sk_buff` header (would have been accomplished in the network stack, had the packet been fully processed via that method)
3. Optionally defragments fragmented packets
4. Populates the timestamp and length fields of the `sk_buff` header section
5. Iterates through a list of all PF_RING ring buffers that currently exist on the system

    (a) For each running ring that is registered against the source device:

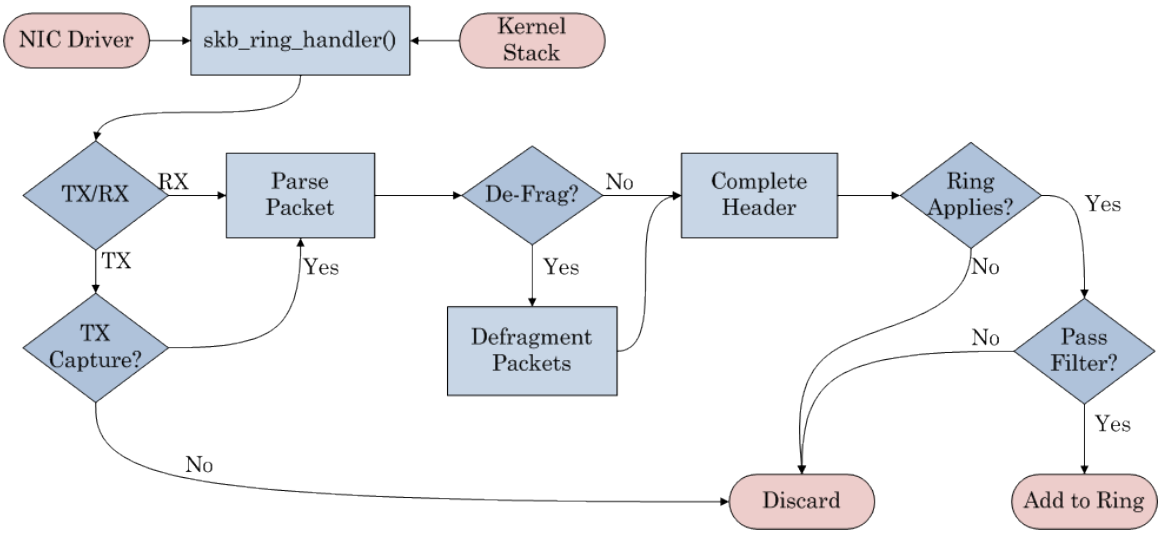


Figure 2.9: PF_RING Protocol Handler Basics: Depicts the basic flow of packets from the NIC to the ring buffer, where it is then the capture application's responsibility to retrieve the packets from the ring

i. The packets are optionally filtered by a rule set attached to the ring

ii. Packets that pass any filters are copied (up to the snaplen configuration of the ring) into an available slot of that ring

(b) If no empty slots are available, the packet is discarded and counted as lost

The PF_RING kernel module is responsible for creating and populating the ring buffer; beyond this, the ring must be consumed by applications using either the libpcap-ring library or the the pfring library directly.

2.3.5.2 The pfring library. The pfring library is used to create, configure, and control ring buffers in the PF_RING kernel module. All actions by the pfring library require the PF_RING kernel module to be loaded. The following list describes the flow of a typical use of the pfring library [14]:

1. Create a PF_RING ring buffer: This is accomplished through standard socket calls, creating a socket in the `PF_RING` domain, with a socket type

of `SOCK_RAW`, and a protocol definition of `htons(ETH_P_ALL)`—which means all packets

```
ring->fd = socket(PF_RING, SOCK_RAW, htons(ETH_P_ALL));
```

2. Configure the ring: Using `setsockopt()`, configure ring settings (i.e., TX-capture, promiscuity, snaplen, application name, etc.)—the below sample sets the snapshot length (snaplen) of the ring slots to capture length (caplen)

```
setsockopt (ring->fd, 0, SO_RING_BUCKET_LEN, &caplen,
                              sizeof(caplen));
```

3. Memory map the ring memory into the application's memory space: Using `mmap()` on the socket descriptor returned when the socket was created, map the portion of kernel memory into that of the user-space application such that the memory is shared between the PF_RING kernel module and the application using the pfring library

```
ring->buffer = (char *)mmap(NULL, memSlotsLen,
                              PROT_READ|PROT_WRITE,
                              MAP_SHARED, ring->fd, 0);
```

4. Enable the ring: Using `setsockopt()` again, instruct the PF_RING kernel module to begin populating the ring

```
setsockopt(ring->fd, 0, SO_ACTIVATE_RING, &dummy,
                              sizeof(dummy));
```

5. Consume the ring slots: In a pthread-safe manner, travel the ring according to standard ring buffer behavior to look for slots with a `slot_state = 1`, representing a populated slot ready to consume and mark empty

   (a) The consumer needs only be concerned with the ring's `remove_idx` and `slot_state`, while the PF_RING kernel module must account

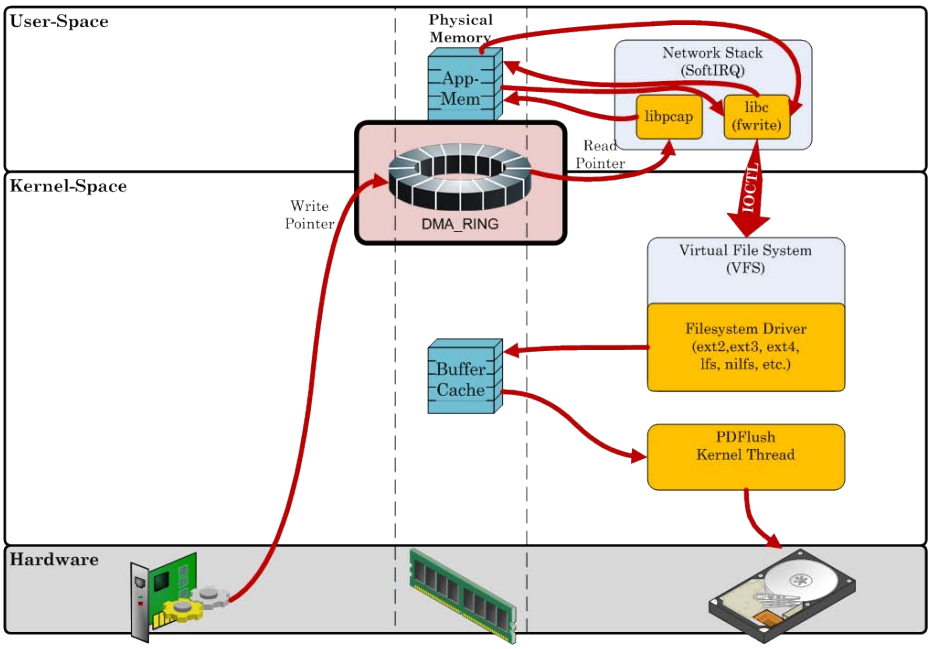


Figure 2.10: The libpcap-DMA-ring Capture Process: replaced driver controls NIC to DMA transfer packets directly to a DMA ring, accessible by a modified libpcap library

for available space and lost packets using both the `insert_idx` and `remove_idx`

(b) The consumer needs to ensure wrapping occurs to maintain a valid index to the ring

6. Closure of the ring: Unmap the shared memory region to shut down ring operations

```
munmap(ring->buffer, ring->slots_info->tot_mem);
```

The pfring library facilitates much greater control and flexibility than the above example. It allows insertion and deletion of filtering rules, creation of ring clusters to load balance captured packets across multiple rings, control of sampling rates for cases where sampling is desirable, and gathering ring statistics [14].

2.3.6 DMA Ring Module. Figure 2.10 depicts the path of packets through a system utilizing a DMA-ring to pass packets from the NIC to the capture application. The method is conceptually similar to both libpcap-mmap and libpcap-ring; however, the research removed another copy by modifying the network device driver and firmware so that the DMA buffer assigned to the NIC was in fact the same ring made available to the capture application. This reduced the copy count by an additional copy, and it allowed for very specific control of the timing from the network device. [8]

This research seems promising for N^{2}d applications; however, the implementation details and source code are not publicly available, and there is no mention of providing filtering logic in the ring. Addition of the filtering logic would require parsing out packet headers, possibly reducing the efficiency and performance of the DMA-ring mechanism as reported by the research team that developed it [8].

2.4 Storage Efficiency

While Section 2.3 describes the advances in capture technology, there is no commensurate treatment of the storage of the captured packets. Filesystem and storage device advances would certainly impact the performance of writing the captured packets to disk; however, these advances would tend to improve the performance of packet storage in the same fashion that improved CPU and bus speeds would improve packet capture. These advances do not address the choke points of the packet workflow.

2.5 Summary

This chapter presents background information on packet capture technology and uses. It lists principle benefits, commonly used tools, and categorized purposes for capturing packets from a live network. The mechanics of capturing packets to a permanent storage device is described, and the technical

challenges of performing N^{2}d capture are discussed. Current research related to improved packet capture technology is presented, and the gap in research for a complete N^{2}d capture system is identified.

III. Design

THIS chapter provides a detailed view of the design and design process of the PKAP kernel-space Nd² capability. Section 3.1 reframes the problem definition in greater detail and describes the aims of the prototype. Section 3.2 describes the challenges and design decisions associated with moving the capture capability to the Linux kernel, and Section 3.3 provides an overview of the resultant architecture.

3.1 Problem Definition

Packet capture systems are computationally disadvantaged in keeping up with network data rates. This is especially true for Nd² systems which are Input/Output (I/O) bound, due to the slow nature of permanent storage. As discussed in Chapter II, many efforts have been applied to improving the performance of getting the packets from the NIC to a user-space program for analysis. This improved performance addresses only one side of the transactions between kernel-space and user-space, while accepting the penalty for the costly inverse copies required to write the captured data to disk.

3.1.1 Goals and Hypothesis. In its simplest form, the objective of this research is to improve the performance of a Nd² capture system. Given this, several obvious approaches are available. Options at the top of the list would certainly include: 1) distribution of the network traffic across multiple systems to keep the load of each machine within the range that allows for reliable capture and storage of network data, 2) increasing system hardware specifications (i.e., CPU speed, Memory size and speed, storage controller, storage media, etc.), or 3) identification of opportunities to utilize the existing hardware more efficiently.

This research targets the last option, since the optimization of code will allow the first two methods to scale more easily to handle even higher net-

work throughput. Regardless of the advances in storage, they will likely never match the computational speed of the processor or communication rates of the network interfaces. The removal of bottlenecks will create the opportunity for improved performance at every level of hardware, allowing users of already-fielded systems to reap the benefit of higher performance without the cost of replacing the existing hardware or of supplementing it with additional hardware.

The following lists the targeted improvements that guiding the design of an improved N^2d capability:

- Reduce CPU utilization for the capture task
- Reduce the memory consumption of the N^2d task as a whole
- Reduce the rate of dropped packets when under high network utilization
- Retain user-space disk responsiveness to satisfy live queries for the captured data

It is no secret that permanent storage is one of the slowest resources of computers and networks today; considering that, the goal of this research could be restated as "enable the N^2d system to store captured packets to disk at the theoretical maximum sequential write speed of whatever permanent storage device is attached, with minimum CPU and memory resourced consumed." Regardless of buffer sizes and speeds, the fastest possible sustainable capture speed is limited to the maximum sustained write speed of the storage device.

It is hypothesized that an N^2d system utilizing a kernel-space capture thread can capture a significantly greater percent of network traffic, with fewer system resources, than a user-space capture application at any network speed. It is also hypothesized that an N^2d capture system can accomplish the improved capture rate without negatively impacting the response delay for live data calls to the disk for the captured data.

3.1.2 Approach. The following sub-sections provide a high-level roadmap in how this research approaches the problem of capturing packets to permanent storage in a disk-bound environment:

3.1.2.1 Solution Space. The first step was to evaluate the chain of custody for the bits from the point the NIC places the received packets in its driver's buffer to the point where they are physically taken out of the buffer cache by the storage controller. It would be ideal to state "the NIC" as the beginning and "the disk" as the end state, but this would be beyond the scope of software programmability–moving into the realm of a hardware solution.

Tracing the bits as they travel through drivers, frameworks, APIs, queues, and applications is not for the faint of heart; and more importantly, it is not possible with closed-source operating systems or drivers. This fact forced the use of an open-source operating system with strong community support. Thus, the basis for the N^{2}d capture system is the 64-bit version of Fedora 11. The open nature of the code and the community provided the best possible opportunity for success. The Linux Cross Referencer (LXR) aided greatly in tracing the packet's path by allowing one to surf through the Linux kernel code in a completely cross-referenced environment.

Further rationale for the selection of a Linux distribution for the operating system is the fact that most software packet capture research available today is on Linux. As seen in Chapter II, the availability of source code, community support, and other research efforts to build on provide strong arguments to use Linux.

3.1.2.2 Direction. Significant performance gains have been achieved through removal of unnecessary layers of abstraction, context switches, and wasteful copies. Positive results from libpcap_mmap, PF_RING, and DMA_RING were beacons pointing the way to remove unnecessary barri-

ers between kernel-space and user-space. The success of these efforts begged the question "could similar gains be achieved by removing the user-space to kernel-space copies when the final destination of the captured data is the disk?"

That question drove the effort to determine the most viable way to remove the copy. Could the downward copy be removed via the mapping of kernel memory into the application space in the same way that the upward copy was eradicated? In theory, the answer was yes; however, further investigation showed that memory mapping was already being used by the user-space libraries to squeeze the most performance out of the constrained interface between user-space and kernel-space. The complexity of attempting to further optimize the existing use of memory mapping, combined with an unchanged penalty for accomplishing the context switches of system calls, pointed to avoiding user-space altogether. The functionality of the capture application is relatively straightforward, and placing it inside the kernel meant that any context switches would be between kernel threads and significantly more efficient.

3.1.2.3 Prototype. The proof of concept for this research is the design, development, and comparative performance analysis of a kernel level N^2d capture capability. The goal of the proof system is to demonstrate that moving both the capture and storage capabilities into kernel-space significantly improves the performance of an N^2d capture system. The performance parameters and the metrics for judging the performance are discussed in detail in Section 4.5; however, it is important to understand that the rate of dropped packets is the most significant internal metric of an N^2d system. CPU utilization, memory utilization, and query response rates are also key metrics—though they mean little if the packet drop rate is high.

3.2 Moving to the Kernel

The PKAP kernel module builds on existing user-space-oriented packet capture research, while adding the ability to store the packets to disk without incurring the penalties for a user-space to kernel-space copy, the context switch, and the overhead involved in a system call. The PKAP-enabled N^{2}d system will retrieve network packets from the NIC, provide filtering capabilities to retrieve only packets of interest, and write the packets of interest to disk in such a format that the network activity can be reconstructed at a later time. These actions are standard N^{2}d activities; whether the application is TCPDUMP, Snort, DaemonLogger, or another custom-built tool to capture packets to disk, each application provides the listed actions as fundamental capabilities.

Hurdles exist for moving an N^{2}d capability into the kernel. The design and development of the PKAP kernel-mode capture capability bears witness that the hurdles are not insurmountable; however, the paths over and around them are not always straightforward or community-supported.

3.2.1 Linux Kernel Modules. Documentation, instruction, and assistance in understanding and implementing Linux kernel modules exist throughout available research, published reference books, and the thriving community of experts on the subject. In fact, one challenge facing new-comers to Linux kernel development is being overwhelmed and confused by the overlapping and oft contradicting material. The Linux kernel has evolved significantly, and its approaches to scheduling, threading, task management, interrupts, network I/O, and countless other areas have changed significantly in just this last decade. Books and papers published for version 2.4 of the Linux kernel can provide pertinent insight into version 2.6 of the kernel for a significant portion of the material; however, it can take inordinate amounts of effort to resolve which information is no longer accurate—or precisely which kernel version makes the information invalid.

A relatively small portion of the Linux kernel, its core functionality, is statically compiled into the kernel binary. A significant percentage of device drivers, non-essential functionality, and other capabilities that are considered more volatile are provided through kernel modules. The modularity of the Linux kernel provides ample opportunity to enhance portions of a running operating system to meet changing demands.

The implementation of the PF_RING protocol handler is one such enhancement. Rather than force changes to other portions of the Linux network stack, the new capability is able to be dynamically inserted without altering the standard behavior of the existing code for other kernel or user-space code that depend on it. Building such capability as a dynamically loadable module prevents the usage of complex conditional logic to determine when to use the new capability. It is employed when the module is inserted, and nonexistent otherwise.

This type of flexibility is necessary for implementing and evaluating kernel-level prototypes; therefore, the design and implementation of the kernel-level N^2d capability is directed to the PKAP kernel module, and the packet capture mechanism to be modified to support network sniffing inside the kernel is directed to the PF_RING kernel module. The ability to dynamically insert these capabilities into a vanilla Linux kernel is important for an accurate performance analysis of the System Under Test (SUT), which will be discussed in greater detail in Chapter IV.

3.2.2 Linux Kernel Threads. Kernel threads are typically used as background processing agents for tasks that may block [13]. One of the most likely examples of such a background process is the `bdflush` kernel daemon. The `bdflush` daemon is responsible for the background task of writing dirty pages from the buffer cache to the physical storage device [2]. While this daemon stood alone at one time, current kernels implement the `bdflush` daemon

as a storage-specific set of the `pdflush` daemon threads, as depicted in Figure 2.7 [38]. The `pdflush` daemon is implemented as a set of threads that both grow and shrink in number within configurable minimum and maximum thread count boundaries to match the demand of storage, and this dynamic thread count will have a particular bearing on both the experimental design and the performance evaluation, to be covered in detail in Chapter IV and Chapter V respectively.

Since any writes to a physical device can block, all kernel capabilities that perform such actions should be implemented either very carefully as a tasklet (currently replacing what was once known as bottom-half code) or as a kernel thread. Tasklets use soft-IRQs and may soon be deprecated in new kernel code due to potential scheduling pitfalls caused by running code at such a high priority [13]. The PKAP N2d function is specifically tasked with writing data to disk—or more accurately, to the buffer cache where the dirty pages will be written to disk by the `pdflush` threads. Therefore, the PKAP module provides the core N2d functionality as a kernel thread.

Several methods exist to create a kernel thread. At this time, the current method of choice for the general case is through the use of the `kthread_run()` helper macro, which will create, daemonize, and wake up the thread [39]. Daemonization of a thread simply means that all file descriptors are removed from the thread (including `stdin`, `stdout`, and `stderr`), all signals are disabled, and the parentage of the thread is changed to `kthreadd`—the default owner of background kernel threads [13]. The PKAP module uses `kthread_run()` to create `pkap_thread`, the daemonized N2d loop.

An additional benefit of implementing PKAP as a kernel thread is that the CPU can switch to "lazy Translation Lookaside Buffer (TLB) mode". The CPU can afford to wait to flush the TLB when executing a kernel thread, since kernel threads have no user-space connections; therefore, kernel threads have

no need to access the user-space page tables. This allows rapid and efficient context switching with respect to the execution of a kernel thread [9].

3.2.3 Capturing Packets from Kernel-Space. Capturing packets from the network is a key function of providing Ned[2] capture services. The various implementations of the libpcap library provide user-space applications ready access to packet capture capabilities; however, no standard libraries exist for capturing packets from within kernel-space.

3.2.3.1 Selection of PF_RING. Several factors went into selecting the method to capture packets inside the kernel. These factors include the following:

1. Efficient
2. High Performance
3. Ability to provide packet filtering
4. Does not require changes to kernel code
5. Can be used for both user-space and kernel-space capture applications in order to control comparative performance analysis to isolate the differences of writing from within the kernel

Considering these factors, PF_RING stands out as a clear choice. Other libpcap implementations require changes to the core kernel code in order to share the ring inside the kernel-space. Development and proper configuration of a `netfilter` plug-in is highly complex and unable to be used by existing user-space capture applications. A custom NIC device driver would be an unknown quantity, and would also be difficult to use with available user-space applications. The DMA-ring would also be a good candidate if source were available for the existing research; however, development of the capture capability itself is beyond the scope of this research. PF_RING provides:

1. Reasonable resource utilization for the given performance [15]
2. Performance above or equivalent to other available libpcap implementations [15]
3. Native packet filtering capability
4. A dynamically loadable kernel module, so the capabilities can be inserted into a standard Linux platform without kernel modifications
5. A lightweight pfring library, demonstrating the essential elements of ring creation, configuration, and use
6. A libpcap-ring library, so existing applications can use the library
7. Multiple sample applications to fully demonstrate library usage

3.2.3.2 Modifying PF_RING. The ring buffer used by PF_RING is memory allocated in kernel space. When used by a user-space application, the pfring library maps the kernel-space memory into the application's memory; however, since `pkap_thread` runs in kernel, only minor modifications to PF_RING are required to support a kernel-space capture thread. These changes include:

1. Exporting a pointer to a PKAP-enabled PF_RING ring buffer

```
#define PKAP_ENABLED


#ifdef PKAP_ENABLED
struct ring_opt *pkap_pfr;
EXPORT_SYMBOL(pkap_pfr);
#endif
```

2. Addition of the `pkap_enabled` flag to `struct ring_op`

```
struct ring_opt {
    u_int8_t ring_active;
    // SNIPPED...
```

```
    /* PKAP Ring Switch */
    u_int8_t pkap_ring; // 0=not_pkap 1=pkap
};
```

3. Addition of `setsockopt()` options to PKAP-enable the initialized PF_RING ring buffer by setting a `pkap_enabled` and pointing the exported pointer to the ring

```
// In linux/pf_ring.h
#define SO_SET_PKAP_RING         119

// In pf_ring.c - ring_setsockopt()
#ifdef PKAP_ENABLED
case SO_SET_PKAP_RING:
    if (pkap_pfr && (pkap_pfr->pkap_ring == 0)) {
        pkap_debug("ERROR: Can only be one PKAP Ring!\n");
        return -EINVAL;
    } else if (pkap_pfr && (pkap_pfr->pkap_ring == 1)) {
        pkap_debug("WARNING: This ring is already a PKAP Ring...\n");
        ret = 0;
    } else {
        pkap_pfr = pfr;  /* This exports the current pfr to kernel
                            space so a PKAP-enable kernel thread can
                            read from this ring through an extern
                            declared pkap_pfr variable */
        pkap_pfr->pkap_ring = 1;
        pkap_debug("Converted PF_RING to PKAP_RING\n");
        ret = 0;
    }
    break;
#endif /* PKAP_ENABLED */
```

4. Addition of `/proc` filesystem status reporting to mark PKAP-enabled rings as such

```
// In pf_ring.c - ring_proc_get_info()
```

```
#ifdef PKAP_ENABLED
    rlen += sprintf(buf + rlen, "PKAP Ring? : %d\n",
                    pfr->pkap_ring?1:0);
#endif
```

5. Altering the `num_ring_users` cleanup facilitator to allow for multiple ring users

```
// In pf_ring.c - add_skb_to_ring()
// Ravelling...
#ifdef PKAP_ENABLED
    atomic_inc(&pfr->num_ring_users);
#else  /* PKAP */
    atomic_set(&pfr->num_ring_users, 1);
#endif /* PKAP */

// Unravelling...
#ifdef PKAP_ENABLED
    atomic_dec(&pfr->num_ring_users);
#else  /* PKAP */
    atomic_set(&pfr->num_ring_users, 0);
#endif /* PKAP */
```

The PF_RING standard functionality and behavior is unchanged from the official PF_RING source until setting `SO_SET_PKAP_RING` with the `setsockopt()` call. This allows the same PF_RING kernel module to service the kernel-space PKAP, the user-space libpcap-ring based application, or even both at the same time (though exercise of that feature is beyond the scope of this research).

3.2.3.3 Accessing PF_RING from PKAP. With a PF_RING capable of accommodating access from within the kernel, the PKAP module requires the functionality of the user-space pflib inside the kernel. Simply using

the user-space library from a kernel module is not possible, so the mechanisms used in user-space must be converted to kernel-space equivalents.

Since the PF_RING module implements the `PF_RING` protocol using the standard Linux networking framework, the pfring uses the standard user-space socket API to create, configure, use, and close a PF_RING ring buffer. The user-space API is executed by syscalls into the Linux core networking code, where a similar kernel-level API exists to carry out the instructions of the user-space calls. The user-space `socket()` function is handled by the kernel-space `sock_create()` function. The functions are roughly equivalent, but the kernel-space functions use a `struct socket` rather than the socket descriptor that the user-space functions use. This is similar to the difference between `struct file` and file descriptors, discussed in 3.2.4.

The PKAP module creates a kernel-space equivalent to every socket API required to use PF_RING. The list of user-space, socket-related calls includes: `socket()`, `bind()`, `setsockopt()`, and `close()`. The kernel-space calls `sock_create()` and `sock_setsockopt()` behave similarly to the user-space calls that share a naming similarity; however, the `bind()` call required a different approach. As shown in the code snippet below, the bind() functionality did not have a kernel-space equivalent (i.e., `sock_bind()`); the socket's `bind()` socket operation had to be used.

```
static int pkap_bind(struct socket *sock) {
    struct sockaddr sa;
    int retval;

    sa.sa_family = PF_RING;
    snprintf(sa.sa_data, sizeof(sa.sa_data), "%s", pkap_device);
    retval = sock->ops->bind(sock,(struct sockaddr*)&sa,sizeof(sa));
    if (retval < 0) {
        printk(KERN_ERR "PKAP: Could not bind socket[%d]\n",retval);
        return retval;
```

```
    }
    debug("Successfully bound socket!\n");
    return 0;
}
```

3.2.4 Breaking the Rules (Writing Files from Kernel-Space). Writing of network data to disk is the second key function of providing N^{2}d capture services; however, this is not a generally-acceptable task for the Linux kernel. The direct access of files from within kernel-space is so taboo, that those who dare request the method to accomplish it are often treated quite harshly. An anecdotal illustration of both the resistance to accessing files from kernel-space and the broad sweeping inclination to do so anyway is the title of an article about such matters: “Driving Me Nuts - Things You Never Should Do in the Kernel” [21].

The reasons for resisting file access from within the kernel are sound. The common purpose for interest in accessing files from a kernel module is to read configuration information from a file. This can be dangerous due to the difficulties in ensuring that the file is available, and that the file contents are as expected. Performing the checks and conditional logic necessary to open, read, and parse a file that could be missing, damaged, or maliciously altered are difficult enough in user-space; however, any error in accomplishing these tasks in kernel-space will likely cause failure or compromise of the system, since kernel modules run at system level without any of the protections or restrictions of user-space applications.

Despite community resistance, methods for accessing files from kernel-space exist in at least two locations [13, 21]. Both sources approach the topic with similar methods for both reading and writing. Though early iterations of the PKAP module employed these methods almost exactly (specifically using `vfs_write()` to write the file data out to the file), the final version directly

```
struct file *log_file;

/* Initial approach */
nbytes = vfs_write(log_file,packet,
                    sf_hdr.caplen + sizeof(sf_hdr),
                    &log_file->f_pos);

/* Final approach */
nbytes = log_file->f_op->write(log_file,
                    bktdata, hdr->caplen,
                    &log_file->f_pos);
```

Listing 3.1: In-Kernel File Write Methods

uses the `write()` file operation pointed to by the in-kernel `file` structure. This method bypasses unnecessary condition checking in the `vfs_write()` call and relies on the `filp_open()` call to ensure that the conditions are met initially, without requiring the checks to be re-accomplished for every write.

Of particular note is that the kernel file operations use `struct file`, rather than the file descriptor that would be returned by a typical call to the user-space `fopen()`. The `file` structure provides the direct access to the file operations for the filesystem that the file resides on. It is this direct access that makes it possible to dispense with `vfs_write()`.

The PKAP kernel module avoids many of the pitfalls of accessing files from within the kernel, since it only writes data to disk. Before writing, PKAP accomplishes the normal file operation checks to ensure the file has been open properly for writing. That said, there is still risk involved in a write-only access from kernel-space. Though the PKAP kernel module never reads the pcap log files it writes, its log file rotation algorithm overwrites files in the course of operation. With no limits on what files can be accessed from kernel-space, misconfiguration or malicious links in the target filesystem could trick the PKAP module into destroying a file or filling a disk that it should not. These potential vulnerabilities can be largely addressed through carefully placed user-space permissions on the target filesystem.

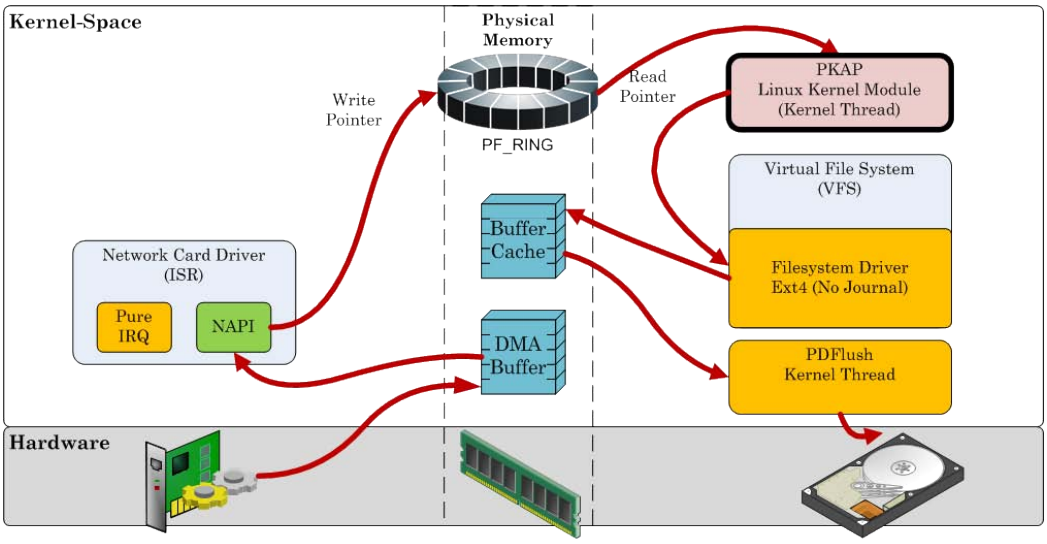


Figure 3.1: The PKAP Capture Process: Using a modified PF_RING, the PKAP kernel module removes all kernel-to-user and user-to-kernel space copying and implements the write() file operation directly without going through the Virtual File System

3.3 Architecture Overview

Section 3.2.3 described the design of the requisite components for a kernel-space N^{2}d system, but this section will provide the architectural overview that describes how the pieces come together. Figure 3.1 depicts the PKAP architecture. Notice that the figure does not contain a user-space area; all packet information and action is handled inside the kernel. The only aspect of the PKAP N^{2}d system in user-space is the `/proc` filesystem status reporting.

Use of PF_RING affords PKAP the same reduction in memory footprint and copy count that it provides user-space applications through pfring and libpcap-ring; however, its use by `pkap_thread` reduces the copy count by one additional copy due to direct use of the ring slot memory for the `file->f_op->write()` call. This behavior differs from an application's use of the user-space pflib library, which copies the data from the ring before it passes the copy to the application. The design decision to directly use the ring slots when executing writes was another change encouraged by the profiling used

during pilot testing. Contrary to user-space writing, the kernel-space file writing was less expensive than the in-kernel `memcpy()` calls used to create copies; therefore, the PKAP module takes every opportunity to avoid `memcpy()` calls, even at the expense of more, and smaller, calls to `write()`. The design decision to forgo buffered `write()` calls is the key difference in memory utilization between a libpcap-based, user-space application and the the PKAP kernel module.

The PKAP main loop, in `pkap_thread`, is very single-minded. It is similar DaemonLogger, a user-space application that is concerned only with retrieving packets of interest from the NIC and writing them to pcap-formatted files on the disk as quickly as possible. The only other task these purist N^2d applications perform is the logic and execution of log file rotation.

As a proof of concept, the PKAP kernel module has no convenient external control mechanisms. All options are currently hard-coded options that require source code alteration and re-compilation. Reporting of the status and health of the ring-buffer is provided by the PF_RING kernel module, and it is this mechanism that provides insight into the status of the `pkap_thread` process. Once key configuration decisions are made in pilot testing, two binary PKAP modules are created: one for saving pcap log files to a real filesystem, and the other for saving the pcap log to `/dev/null`.

IV. Methodology

THIS chapter presents the methodology used to evaluate the performance of an N^{2}d Network Capture System using the PKAP kernel module compared to traditional user-space applications. The evaluation will use 4 metrics: CPU utilization, memory usage, dropped-packet rates, and query response delay. Section 4.1 charts out the system boundaries, and Section 4.2 presents the evaluation methodology for the system. Section 4.3 discusses the services that are provided by the N^{2}d system, as well as when those services are considered successful. The workload types and descriptions are presented in Section 4.4, and Sections 4.6 and 4.7 discuss the system parameters, as well as which ones will be treated as factors during the evaluation. The remaining sections of this chapter, Sections 4.8 and 4.9, explains both the evaluation technique and experiment designs used to test, analyze, and interpret the system performance.

4.1 System Boundaries

As depicted in Figure 4.1, the SUT is the N^{2}d Capture System. Though the N^{2}d capture system provides a relatively straightforward capability, being derived from a general purpose, multitasking, multiprocessing computer system means that there are are large number of elements to the system of systems. Therefore, Figure 4.1 will show only those components that have the most direct bearing on the function and performance of the N^{2}d tasks.

The key N^{2}d capture system components are the capture method, the capture library, the NIC and associated driver, the physical storage type and speed, the filesystem, and the agent used to retrieve specific captured data at the behest of external queries. The Component Under Test (CUT) is the capture method, which is comprised of the interchangeable capture tools: PKAP and DaemonLogger. The system will be evaluated under specific variations in workload and factor manipulation for each CUT option separately.

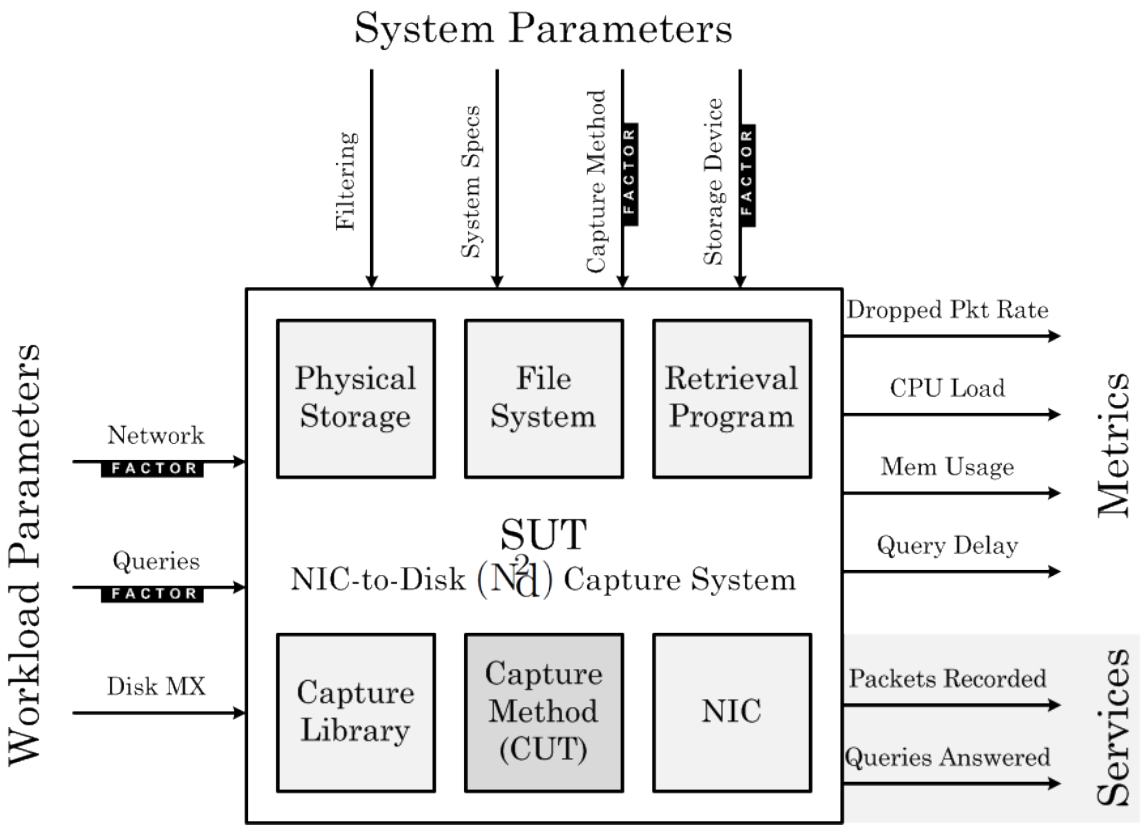


Figure 4.1: Systems Approach Diagram for the NIC-to-Disk Capture System

Workload parameters include the network traffic, external queries for captured data, and the disk maintenance of log file creation, rotation, and removal. The system parameters include the storage device type, packet filter count, and capture method (the CUT). Additional parameter details are found in Section 4.6.

The metrics used in analysis of the system performance are packet drop rate, CPU load, memory usage, and query delay. Ultimately, the most important metrics are packet drop rate and query delay. The rationale for the metrics and the methods used to obtain and interpret them are covered in greater detail in Section 4.5.

4.2 Evaluation Methodology

Measurement of a real N^{2}d system is used to accomplish the performance evaluation during the experiments. The measured results are validated using analytic analysis. As the N^{2}d capture system is a highly interdependent system of systems, accurate modeling of the system behavior and performance characteristics would be extremely complex. Additionally, a model would require significant maintenance to account for different hardware or software. Similar difficulties would exist for simulation of the systems many interrelated subsystems.

To evaluate the performance characteristics of a PKAP-enabled N^{2}d system, actual performance metrics must be obtained for both the kernel-space capture capability and an equivalent user-space capability. Representing the user-space *status quo*, is DaemonLogger—a tool written for the express purpose of capturing network packets to disk. DaemonLogger uses libpcap to capture packets and write them to disk in pcap-formatted files, and it manages a configurable rotation of the files on disk for long-term collection. DaemonLogger typically uses the standard libpcap library; however, for purposes of this evaluation, it was linked with the PF_RING-enabled libpcap.

The PKAP kernel module was implemented to write files in the pcap-format and provide a similar functionality for rotating the capture logs. The primary functional distinction between DaemonLogger and PKAP is the memory space in which they run. Both will utilize the PF_RING ring buffer, so the efficiency of the capturing process itself should be identical up to the point where the capture application actually removes the packet element from the PF_RING ring buffer. Both applications then prepend a pcap-format packet header to each packet and write them to disk—PKAP using the kernel's `vfs_write()` function, and DaemonLogger using the glibc function `fwrite()`. The `fwrite()` function uses buffering by default, since generating

fewer system calls with large chunks of data is more efficient than many system calls with small amounts of data each time. This buffering behavior was not implemented in the PKAP module. Though this is another distinction between the methods, the justification for the difference lies in 1) the goal to use less memory, and 2) the relatively low cost of calling `vfs_write` from within the kernel itself versus across the system call boundary.

Additionally, to ensure the DaemonLogger application could be used as a control in all trials, slight modifications were made to it to allow it to write to /dev/null. This option was not available in the original DaemonLogger application, nor would the file naming conventions permit filesystem trickery to force it to work without modifying the application.

4.3 System Services

The N^{2}d capture system provides both internal and external services. The ultimate utility of the system is that any given subset of stored packets can be requested from the system and retrieved because they have been accurately recorded from the network to the system's storage device. For this external service to be successful, the N^{2}d system must first provide the internal service of accurately storing all packets of interest to disk in the proper format.

At a high level, the system is considered to be successful at providing service if it:

1. Accurately stores packets of interest while incurring minimal losses due to dropping packets during the capture process
2. Provides adequate processing and disk I/O to respond to queries for the captured date in a timely manner

This ideal view is useful when considering the system as a whole; however, the disk-bound nature of packet capture speed requires a more granular concept of success. The internal service is considered successful when network traffic

can be captured to disk at the maximum write speed of the storage device. Due to the additional Bytes used to prepend the pcap-format packet header to each captured packet, the data rate will be slightly less to disk than from the network. This disparity between network data rates and sequential write data rates is an inverse power relationship—meaning the smallest packets carry the greatest data rate overhead when writing to disk. The worst case scenario when 64 Byte packets are filling the network, which requires the storage device to be capable of writing 130 MB/s to disk for every 100 MB/s received by the NIC.

Given an N^2_d capture system with a storage device that can sustain a sequential write speed of 70 MB/s on a network where the average packet size is 768 Bytes, the maximum theoretical data rate the host system could receive before dropping packets would be 545.6 Mbps (68.2 MB/s). The theoretical data rate would be reduced to 424 Mbps (53 MB/s) if the average packet size is 64 Bytes.

Additionally, the maximum theoretical sequential write speed is only applicable when no other read or write operations are being accomplished on the storage device. Queries for data will impact performance of the recording capability by introducing disk I/O contention.

The choice of storage device will greatly impact the ability of the system to be successful at a given network load.

4.4 System Workload

The workload of the SUT consists of the network traffic, the queries for data, and the disk maintenance overhead for packet log rotation. The network traffic characteristics is varied by both size and data rate during the performance evaluation, and the query workload is only introduced in the final experiment. The disk maintenance workload is consistent between CUT options

and held to the initial configuration throughout all experiments after pilot testing.

4.4.1 Network Traffic.

4.4.1.1 Characteristics. Network traffic is the principle workload for an N^{2}d system. The size and frequency of packets determine the ultimate stress of the packet capture, storage, and retrieval functions. The network workload is a synthetic workload produced by the `pktgen` Linux kernel module. Pktgen is a configurable User Datagram Protocol (UDP) bit-spitter which was designed to test the TX and RX capabilities of NICs and their associated drivers. This method of traffic generation is sufficient for testing N^{2}d systems for the following reasons:

1. The packet payload is not analyzed, so payload and protocol details above L4 only matter for packet size
2. The PF_RING mechanism removed the `sk_buff` from the kernel TCP/IP stack prior to the point where a distinction would be made between the way a given protocol is handled
3. The filtering capability used in the PF_RING is a simple Access Control List (ACL), where the imposed load of execution is not affected by state or protocol type

As seen in Figure 4.2, the traffic generator is run on a test machine connected via a single Gigabit Ethernet port. This setup can saturate a Gigabit Ethernet link until the average packet size drops below 700 Bytes. Multiple pktgen systems are required to fully saturate a Gigabit Ethernet link with small packets. A single traffic generation system should produce sufficient traffic types and rates for initial evaluation of the PKAP kernel mode capture capability, since the N^{2}d test system stores packets to a single Serial Advanced Technology Attachment (SATA) II hard drive with a maximum sequential block write speed

of 82 MB/s, as determined by the `bonnie++` utility—a standard user-space test tool for hard drive performance. This maximum speed is considered to be the highest guaranteed speed that a user-space application can stream data to disk sequentially. Additional traffic generation capabilities would be required if the test system utilized high-end storage devices, such as SANs, Network Attached Storage devices (NASs), or directly attached Redundant Array of Innexpensive Disks (RAID) arrays.

To test beyond the limited data rate of a single SATA disk, a portion of the testing scenarios will be repeated while writing to `/dev/null`. This will essentially remove the bottleneck of the available hardware storage devices to determine comparable performance and/or failure levels at extremely high data rates.

The network workload is varied in both size and data rate to exercise the chain of custody for the bits from the NIC to the disk. While bitrate is a significant factor in the network workload, the rate of packets per second is actually more significant to CPU utilization than the bitrate of the network—specifically, higher quantities of small packets filling a given bitrate provide significantly greater workload than lower quantities of large packets filling the same bitrate. Three size setting are used to evaluate behavior of the CUT: min, max, and random. Min test sets transmit only 64 Byte frames to evaluate how the CUT impacts performance of the SUT when the chain of custody spends more time setting up the handling of a packet buffer than actually transferring from one memory location to another. Max test sets transmit only 1500 Byte frames to evaluate the CUT impact on the SUT when the principle task of the system is to move the packet data, with minimum resources spent on setting up the movement. Random test sets transmit a normal distribution of packet sizes, ranging from 64 Byte to 1500 Byte frames. The random frame sizes produce an average frame size of 782 Bytes, and the single pktgen generator can fill the network link to 950 Mbps with this setting. The purpose is to evaluate the

performance of the system when the principle load of capturing the packets is balanced between the overhead of setting up packet movement and the actual copying of data from one memory location to another.

4.4.1.2 Bitrates. The data rate of packets can be altered specifically by adapting the pktgen delay. This nanosecond-level adjustment of delay between packets alters the throughput of the network by imposing an artificial Inter-Packet Gap (IPG) on the network link. The effect of the IPG on the data rate is moderately accurate. It would be more accurate if a real-time Linux kernel with high resolution timer were utilized; however, the subtle variations of 2-5% in transmission rates are acceptable. Additionally, since each test configuration will be accomplished five times, the variation could provide a more representative understanding of CUT performance under real-life workloads.

4.4.1.3 Duration. Pilot tests reveal that trials of 5 minutes are sufficient to bring the SUT to a steady state at even the lowest trial bitrates. Bringing the system to the steady state is important for valid analysis of the collected metrics. The steady state refers to bringing the N^2d capture system to the point where the temporal surge capability provided by the Linux buffer cache is not a factor in the performance of the system and the portions of the data required to satisfy the queries for data are not likely to be able to reside in the buffer cache. If the buffer cache can satisfy a significant portion of the storage or query tasks, the resultant data value is less likely to represent real-world performance of long-term N^2d capture services.

4.4.2 Data Queries. The workload provided by data queries requires both the creation of distinguishable packets to search for and the query agent to accomplish the search and response.

4.4.2.1 Query Object Creation. The query objects (marked packets) are transmitted over the network with specific address ranges unique to the query workload. These specially addressed packets are sent once every second so that they are interspersed within the general network traffic. In and of themselves, these marked packets provide no noticeable workload increase to the system; however, the SUT must search for these marked packets in the packet data it has stored. This workload is considerable given that trials produce between 4 GB and 40 GB of network data on disk. The marked packets are identical to the general background traffic in all ways but the Internet Protocol (IP) addresses.

4.4.2.2 Query-Response Workload. To represent the external requests for data that an N^2D system would receive in a real-world situation, the SUT runs a query script that initiates periodic queries for packet data. This script uses TCPDUMP to look through all captured data for the marked packets and to copy the packets to a temporary query response file.

The impact of the query workload to the system is multifaceted, in that the SUT CPU resources are being consumed by an additional task, and the storage device can no longer be dedicated to writing alone. The disk must satisfy both the write and read demands, significantly impacting both the write performance and dropped packet rate due to contention of disk I/O.

4.4.3 Disk Maintenance. The principle component of the disk maintenance is the log rotation performed by the CUT. Regardless of whether the test CUT is the PKAP module or the DaemonLogger application, the log rotation logic is configured to behave identically. This configuration allows each CUT to create pcap log files up to 1 GB in size, after which a new log file is created. The CUTs are configured to keep ten log files. Once the tenth log file fills, the oldest 1 GB file is deleted and a now-10th file is created. File creation and deletion

provide a periodic workload for the SUT filesystem. Though the workload is not precisely measurable, the consistent configuration of both CUTs ensures that the workload is the same throughout testing. This allows disk maintenance to be a parameter (not a factor) during performance analysis.

4.5 System Performance Metrics

The N^2 system must provide reliable answers to queries for captured data with a high probability of success. For this to occur, the system must capture packets and stored them to disk with the lowest probability of dropped packets. To be successful at both capturing the packets and providing timely responses to queries for the data, the N^2 system must operate at the highest possible efficiency—processing packets from the NIC to the disk with as little overhead as possible. To evaluate whether the PKAP CUT method provides a higher probability of satisfying the system services, the following metrics are defined:

- Dropped Packet Rate
- CPU Utilization
- Memory Utilization
- Response Delay for Data Query Responses

4.6 System Parameters

System parameters are the properties of the SUT that will affect its performance. As a system of systems built on a general purpose operating system, the complete list of parameters would be staggering. Utilizing subject matter expertise and the initial findings during the first phase of experiments, the list was narrowed down to those things which will most significantly affect the performance of the SUT's ability to perform its services. System performance is also affected by workload parameters. The following sections list and describe the system parameters for the SUT.

4.6.1 Capture Method. The CUT selection determines whether the packets must pass through user space or can remain in the kernel during their conveyance from the NIC to the disk inside the SUT. Selection options are: 1) an existing user-space capture capability, DaemonLogger, or 2) the proof of concept produced in the course of this research, the PKAP kernel module. Both methods use the same capture library to both retrieve packets from the NIC and filter them for interest, and both methods store to the same disk configuration using the Linux kernel's Virtual Filesystem Service (VFS). The differentiation between CUTs is the location of the capture logic in the SUT.

4.6.2 Disk Device Selection. The disk configuration greatly impacts the performance of the SUT to successfully provide its services. The disk configuration directly affects the theoretical limit of network capture, since an N^{2}d system can only sustainably capture network traffic at a data rate commensurate with the maximum block sequential write speed of the storage device being written to. During the evaluation, the storage device options will be 1) a single SATA II hard drive formatted to Extended Filesystem version 4 (ext4) and capable of 65 MB/s, or 2) the `/dev/null` device.

The ext4 file system was selected due to existing benchmarks that list ext4 sequential write speeds as superior to all other current Linux filesystem [23]. This selection is verifiable using the `bonnie++` hard drive performance test utility. During recent tests of Linux filesystem performance, the ext4 filesystem bested ext3, XFS, and ReiserFS by 10-40% in large file streaming writes [23].

4.6.3 Filtering. Filtering is a system parameter for an N^{2}d system. The filtering is the intentional packet discarding by pre-configured rule set. The SUT is capable of filtering packets, regardless of CUT selection. As stated in Chapter IV, the ability to perform filtering was a key factor of the design process; however, all filtering is disabled during this performance analysis. The

filtering process behaves the same for both CUT selections and provides no insight into the impact of the CUT selection on SUT performance.

4.6.4 System Specifications. The hardware specifications for the host system impacts the performance of the Nd^2 capture application. Increased performance of the CPU, memory, storage system, and NIC can all directly influence the performance of the SUT; however, these changes in performance are outside the scope of this research. The specifications for the test system are held constant throughout testing.

4.7 System Factors

This section presents the selection of system and workload parameters to be used as factors in during the evaluation. These factors were selected through a combination of expert knowledge and pilot studies. Factors are varied differently during the three experiments. Experiments 1, 2, and 3 are detailed in Section 4.9.

The factors in Experiment 1 are varied to compare the impact of CUT selections on the performance of the SUT under controlled workload and system configurations using a real storage device and requiring no queries for captured data. The factors in Experiment 2 are varied similarly to Experiment 1; however, the virtual device `/dev/null` is used in the attempt to observe the behavior of the CUT selection when disk I/O is not a factor. The factors in Experiment 3 are varied to determine the impact of the query workload on capture performance, which requires the captured packets to be stored to a real storage device. Sections 4.7.1 to 4.7.2 outline the factors selected from the system and workload parameters and each of the levels used during the experiments.

4.7.1 Capture Method. The capture method is the CUT. The capture method determines whether the packets must traverse user-space or not when

capturing packets to disk, and the analysis of the impact of this factor is the principle goal of this thesis.

4.7.1.1 PKAP Kernel Module. The PKAP kernel module is the proof of concept for a kernel level N^{2}d capture capability. PKAP is a dynamically loadable Linux kernel module that implements a background thread. Greater detail of the PKAP design is in Chapter III.

4.7.1.2 DaemonLogger. DaemonLogger serves as a baseline functionality and performance threshold for user-space N^{2}d capabilities. The DaemonLogger application is a user-space, libpcap-based packet capture application written by Martin Roesch—author of the Snort Intrusion Detection System. DaemonLogger provides a very efficient and purposeful N^{2}d functionality, and nothing else. The straightforward nature of the application made it a strong candidate for representing a user-space capture application.

DaemonLogger was compiled against the libpcap-1.0.0-ring library, which is the PF_RING-enabled version of the standard libpcap library almost universally used to implement user-space network sniffing applications. Since the DaemonLogger application retrieves the packets from the same ring buffer implementation as the PKAP kernel module, the retrieval process is identical up to the point of the CUT. Additionally, DaemonLogger uses the `fwrite()` call (via libpcap's `pcap_dump()` function); `fwrite()` uses the `write()` syscall, which is handled by the `vfs_write()` function in the Linux VFS. The VFS transparently passes handling of this call to the filesystem's `write()` file operation. As discussed in Section 3.2, the PKAP kernel module uses the `write()` file operation directly. This guarantees that the storage process is identical from the point that the packet data is passed to the kernel's filesystem `write()` file operation to when it is written to disk. Due to careful selection of application and configuration, the principle difference between CUT selections is iso-

lated to how the application calls the write file operation of the filesystem the log file is stored in.

4.7.2 Storage Device. Most trials will be accomplished using the real hard drive in the N^{2}d system; however, to extend testing beyond the ability of a single SATA II hard drive sequential write speed, certain trials will use the `/dev/null` device. The details of the storage device options are as follows:

- Real Disk:
  - SATA II
  - 7200 RPM
  - 16 MB buffer
  - Connected to an Intel® ICH9M-E SATA RAID Controller
  - Bonnie++ tested to have a maximum sequential block writing rate of 65 MB/s
- `/dev/null`
  - Virtual device on Linux that:
    * Immediately discards all data written to it
    * Always reports SUCCESS for any write
    * Always reports End Of File (EOF) for any read
  - An educated trial indicates that the performance of the /dev/null device on the N^{2}d system is capable of a sequential block writing rate between 2.6 GB/s and ∞
    * No known benchmarking utilities offer the ability to test a device that can't be read from
    * The upper bound was found by timing the copy of 4 GB from `/dev/zero` to `/dev/null` using `dd`

- `# time dd if=/dev/zero of=/dev/null bs=4096 count=1048576`
- The result from `time` was 0 seconds

* The lower bound was found using the command `# hdparm -t /dev/zero`
  - It is assumed that if the VFS can read a CPU-bound virtual device at this rate, then it is likely that the VFS can write to a CPU-bound virtual device at a similar rate

– Trials using `/dev/null` cannot be verified nor queried for, since the data is immediately discarded

4.7.3 Network Workload.

4.7.3.1 Packet Size. Though the key service of the N^{2}d system is to provide the requested packet data to the requesting party, the most significant workload of the system is the copying and formatting of packet data from the NIC to the storage device. Packet size is one of the most significant contributors to the stress of a system—potentially even more so than the bitrate. Network infrastructure and devices typically use 64 Byte packets for stress testing. Though a network pipe filled with nothing but 64 Byte packets is not likely to be found in operational network use, it provides a worst-case scenario that network developers need to ensure proper functionality during strange and unknown circumstances in the field. During the experiments the packets sizes will be controlled to be one of the following:

- Min Size: 64 Byte frames
- Max Size: 1500 Byte frames
- Random Size: A randomly selected, normal distribution of packet sizes ranging from 64 to 1500 Bytes

4.7.3.2 Bitrate. As with packet size, the bitrate of the traffic is one of the two most significant workload sources for the SUT. In order to evaluate the system in times of low stress, tipping point stress, and theoretically unlimited stress, the bitrate will be controlled to the following levels:

- Level 1 Bitrate: 100 Mbps (~12 MB/s): Low end testing that to explore the CUT impact on the SUT when the load is light
- Level 2 Bitrate:
  - [64 Byte Packet Size] 140 Mbps (~18 MB/s): below the data rate of the hard drive, but close to the processing limits for the packet processing routines to handle the packets per second (pps) associated with this bitrate
  - [1500 Byte and Random Packet Sizes] 650 Mbps (~82 MB/s): right at the maximum data rate for the hard drive in the test N^2d system, as determined by pilot testing
- Level 3 Bitrate:
  - [64 Byte Packet Size] 200 Mbps (~25 MB/s): below the data rate of the hard drive, but moderately above the processing limits for the packet processing routines to handle the pps associated with this bitrate
  - [1500 Byte and Random Packet Sizes] 700 Mbps (~88 MB/s): moderately above the maximum data rate for the hard drive in the test N^2d system, as determined by pilot testing
- Level 4 Bitrate:
  - [64 Byte Packet Size] 340 Mbps (~43 MB/s): below the data rate of the hard drive, but well beyond the processing limits for the packet processing routines to handle the pps associated with this bitrate (the pps rate is at the maximum for the switch in the test environment, and it requires 2 pktgen systems to create)

- [1500 Byte and Random Packet Sizes] 980 Mbps (~120 MB/s): well beyond the maximum data rate for the hard drive in the test N2d system

Experiment 3 is intended to more closely represents the real-world workload of an N2d system, so the packet sizes are randomly selected between 64 and 1500 Bytes. The distribution of the packet size selection is normal, and the average packet size is 782 Bytes; thus, the network workload levels for experiment 3 are categorized by bitrate alone—not packet size. The bitrate levels for experiment 3 are selected to provide increased granularity at lower network bandwidths to determine the effect of the query workload on the SUT. The following are the bitrate levels for experiment 3:

- Level 1 Bitrate: 100 Mbps (~12 MB/s)
- Level 2 Bitrate: 225 Mbps (~28 MB/s)
- Level 3 Bitrate: 350 Mbps (~32 MB/s)
- Level 4 Bitrate: 575 Mbps (~72 MB/s)
- Level 5 Bitrate: 650 Mbps (~82 MB/s)
- Level 6 Bitrate: 700 Mbps (~88 MB/s)
- Level 7 Bitrate: 975 Mbps (~122 MB/s)

For all experiments, network speeds producing storage speed requirements beyond the capacity of the storage device provides insight into the failure mode characteristics of the CUT.

4.7.4 Query Workload. The query workload is either applied or absent within a given trial. The query workload rate is not altered for trials that include a query workload.

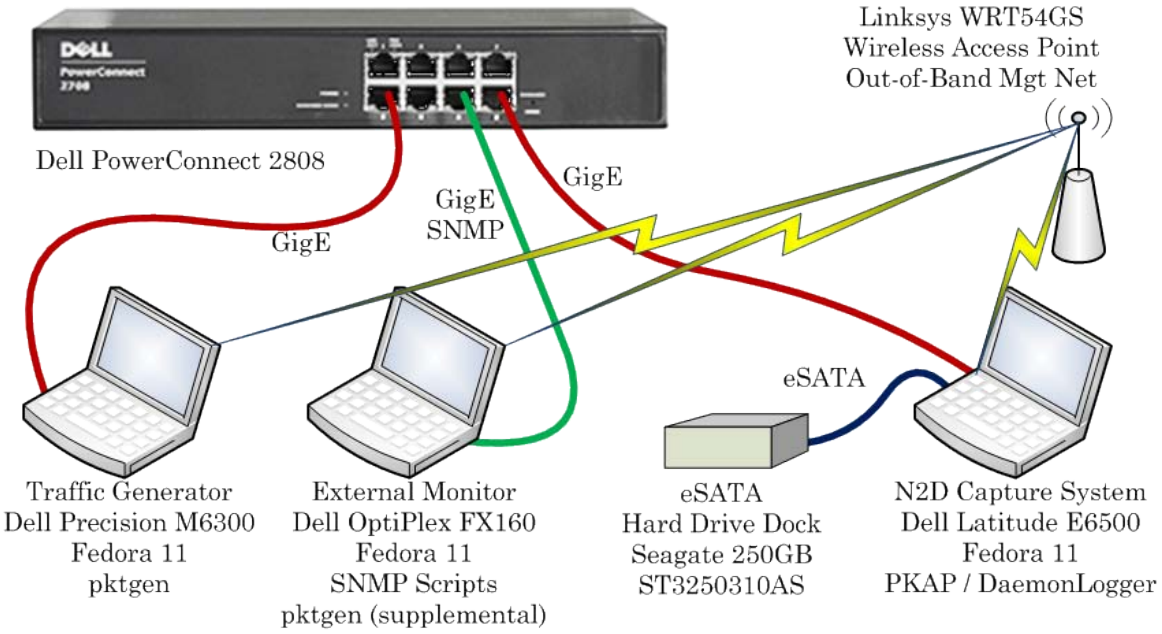


Figure 4.2: Diagram of the Evaluation Environment

4.8 Evaluation Technique and Environment

Evaluation of the SUT is conducted through the measurement of a real N2D capture system in an environment that is representative of real-life network traffic. Analytical Modeling and Simulation for the system-of-systems would be both complex and unreliable. Additionally, the development of the PKAP proof of concept provides greater flexibility to adapt the functionality to new environments or platforms.

4.8.1 Evaluation Environment. As depicted in Figure 4.2, the environment used to evaluate the SUT is a small network comprised of the following devices and functionality:

- Dell PowerConnect 2808 Managed Gigabit Switch: An 8 port Gigabit switch, configured to provide interface statistics via Simple Network Management Protocol (SNMP) to software on the monitoring system to provide hardware packet counts and rates. This switch provides the test network data path.

- Linksys WRT54GS Wireless Router: This wireless access point provided the out-of-band control network for the test environment, allowing the automation scripts to orchestrate each test run without impacting the test network data.
- Traffic Generator: The Dell Precision M6300 runs Fedora 11 and the pktgen kernel module for generating traffic. The system has a Gigabit Ethernet port and an Intel® Core™2 Duo T9500 @ 2.6GHz, and it is capable of saturating the 1 Gigabit per second (Gbps) connection when the average packet size is sufficient (above 700 Bytes)
- External Monitoring System: The Dell Optiplex FX160 runs Fedora 11 as the operating system, and it runs the Net-SNMP scripts to monitor the interface statistics of the Gigabit Switch; these statistics will be used to validate the network statistics from the Traffic Generator and SUT. Additionally, the monitor system is tasked as a supplemental pktgen traffic generator—used to boost the bitrate of the 64 Byte packet, 340 Mbps trial, and to send the specially marked packets used to provide the query workload. This system also runs a Network Time Protocol (NTP) server to provide a synchronized clock between all test systems. The synchronized clock allows data gathered by each system to be compiled together; which, in turn, enables batches of test sets to be scheduled in long runs to satisfy the significant number of iterations that are required to evaluate the SUT under the levels of factors listed in Section 4.7.
- N^2d Capture System (the SUT): The Dell Latitude E6500 runs Fedora 11 and the CUT. Depending on the trial, the CUT will either be the PKAP kernel module or the DaemonLogger user-space service. The pertinent system specifications are as follows:
  - Intel® e1000-based Gigabit Ethernet adapter
  - Intel® Core™2 Duo T9600 @ 2.8GHz

- 8 GB RAM
- Internal 2.5 inch SATA System Drive
- External 3.5 inch External Serial Advanced Technology Attachement (eSATA) Capture Drive (Seagate ST3500620AS)
- Fedora Core 11 (2.6.30.10-105.fc11.x86_64 Symmetric Multiprocessing (SMP))
- PF_RING version 4.1.0 Revision 3982 (plus the modifications to share memory inside the kernel with PKAP)
- libpcap-1.0.0-ring (distributed with PF_RING 4.1.0)
- PKAP kernel module version 1.0.0

4.8.2 Evaluation Technique. This section describes the methods used to collect the data from each trial and the process used to analyze the data. The metrics are used to compare the performance of the new design to that of the traditional approach—that is, to compare the affect of the PKAP kernel module to the SUT performance with the DaemonLogger application. Each trial set for the experiments discussed in Section 4.9 provides a specified factor level combination. Each combination of factors presents a load-level for which the metrics between one CUT option can be directly compared to the other.

4.8.2.1 Dropped Packet Rate. Though not exclusively, the packet drop rate is one of the most significant metrics for an N^2d system. Any success in the remaining metrics are largely voided if the probability of successful packet capture to disk is decreased. During each trial, the traffic generator will send sufficient packets to ensure that the trial duration is at least 300 seconds (approximately 5 minutes). The packet drop rate is related to the probability of successful packet capture by the following:

$$p(drop) = 1 - P(capture),\ where\, P(capture) = \frac{Number\, of\, Packets\, Stored}{Number\, of\, Packets\, Transmitted} \tag{4.1}$$

The packet drop rate is measured by comparing the number of packets transmitted by pktgen to the number of packets stored to disk by the CUT. Following the matrix of test parameters, each test set will be run 5 times. The number of trials for each test configuration is initially established during pilot testing. The SUT is a system of systems, based on a general purpose operating system; multiple trials are required to provide a reasonable confidence interval during slight variations of overall system resource utilization. Pilot testing indicates that 5 trials per test configuration is sufficient, and that greater than 5 trials provide no reduction of the 95% confidence interval range.

4.8.2.2 CPU Utilization. The N2d system load is directly impacted by the network workload, and this data will be captured during each trial for comparisons to be made between CUT options. CPU utilization is gathered by the `pidstat` utility, which is part of the systat suite of system status monitoring tools. The `pidstat` command will be configured to track the workload of the CUT process or kernel thread, collecting the data every 5 seconds for the duration of the test.

The average CPU utilization for a given set of factors is the primary CPU metric of interest; however, anecdotal use of the changes over time could assist in determining the source of other potential bottlenecks, changes in traffic, unexpected background tasks, etc..

4.8.2.3 Memory Utilization. Memory utilization is tracked using the same methods as CPU utilization.

Table 4.1: Table of Factor Configuration Sets: Listing each configuration set for Phases 1-3, all sets contain 5 trials for each CUT option

P1	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12
Pkt Sz	min	max	rnd	min	max	rnd	min	max	rnd	min	max	rnd
Mbps	100	100	100	140	652	650	200	714	700	340	985	975

P2	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12
Pkt Sz	min	max	rnd	min	max	rnd	min	max	rnd	min	max	rnd
Mbps	100	100	100	140	652	650	200	714	700	340	985	975

P3	S1	S2	S3	S4	S5	S6	S7
Pkt Sz	rnd	rnd	rnd	rnd	rnd	rnd	rnd
Mbps	100	225	350	575	650	700	975

4.8.2.4 Query Delay. During the Experiment 3, a reduced factor set is used to run the CUT options against both network and query workloads. The above metrics continue to be used; however, metrics for the query-response cycle are additionally monitored and analyzed. The query-response metrics of interest are: the query-response cycle delay in seconds, the number of queries successfully completed, and the percentage of the marked packets found.

4.9 Experiment Design

There are four experimental phases. The first phase is pilot testing, with the goal of determining fixed parameters, variable factors, data collection methods, and network infrastructure reliability and settings. The second phase is focused on determining the performance of CUT selections under typical and atypical loads while providing the service of N_d^2 full-packet capture to a real storage device. The third phase is executed similarly to the first phase; however, the real storage device is replaced by the `/dev/null` virtual device. The third phase uses typical and atypical loads to determine the performance of CUT selection when the SUT is providing the second service—response to calls for data.

4.9.1 Experiment 0: Pilot Testing. This section describes the pilot testing purposes and methods. The pilot testing is useful during development of both the CUT options as well as the test environment.

- Configurations: Series of tests are conducted to determine reasonable configuration settings for PF_RING slot count and filesystem formatting and formatting options. These tests events assisted in proper configuration of the SUT and test environment.
- Network Architecture: Evaluations of several packet generation technologies are evaluated to ensure that the system can be fully exercised and that the test environment can be both controlled and monitored. Key decisions include: determining whether a switch or cross-over cable would be used, whether testbed control would occur in-band or out-of band, and which hardware platforms of those available would best fulfill the duties of the SUT, traffic generator, and monitor.
- Traffic Generation: Series of tests are conducted to determine configuration options of the packet generation tool to ensure that the process is controllable, reliable, and repeatable. Establish a configuration file format to identify traffic generation speeds, sizes, addresses, and duration for each given trial set.
- PKAP Profiling: Once the testbed is capable of generating network traffic, trial runs are used to provide a workload to profile the PKAP kernel module code using the `oprofile` test suite. This is accomplished toward the end of the prototype development phase.
- Measurement Tools: Once the testbed is capable of generating network traffic, many iterations of trial runs are utilized to assist in evaluation of measurement tools to acquire data from each of the systems on the testbed. These tools include and scripts utilities to monitor the CUT process metrics, the status of the PF_RING ring buffer health and drop

count, the packets transmitted by the traffic generator, and the packets confirmed by the switch to have been sent to the SUT. All activity logged must be accompanied by a timestamp that is based on a synchronized clock between all three testbed systems.

- Trial Determination: Cursory tests are accomplished to identify:
  - A rough estimate of the variability of the results to determine the initial trial count
  - Applicable bitrate levels to observe requisite characteristics of the SUT
  - Duration of trials to ensure SUT has reached a steady-state

4.9.2 Experiment 1: Packets to Disk. As depicted in Table 4.1, Experiment 1 has 24 distinct test configurations (2 CUT options * 12 Trial Sets). The 24 test configurations each receive sufficient packets to bring the capture system into a steady state for the trial period. Each trial set contains 5 independently accomplished trial runs for each CUT, using the same SUT and environment configurations. For Experiment 1, the testbed produces, transmits, receives, and writes to disk roughly 6.7 billion packets carrying nearly 2.5 TB of network traffic over the course of 10 hours.

4.9.3 Experiment 2: Packets to `/dev/null`. As depicted in Table 4.1, Experiment 2 has 24 distinct test configurations (2 CUT options * 12 Factor Configuration Sets). The only change between Experiment 1 and Experiment 2 is that the storage destination of the packets is `/dev/null`, rather than a real disk. As in Experiment 1, each trial set contains 5 independently accomplished trial runs for each CUT, using the same SUT and environment configurations. For Experiment 2, the testbed produces, transmits, receives, and writes to disk roughly 6.7 billion packets carrying nearly 2.5 TB of network traffic over the course of 10 hours.

4.9.4 Experiment 3: Impact of Data Queries. As depicted in Table 4.1, Experiment 3 has 14 distinct test configurations (2 CUT options * 7 Trial Sets). The 14 test configurations each receive sufficient packets to bring the capture system into a steady state for the trial period. Each trial set contains 5 independently accomplished trial runs for each CUT, using the same SUT and environment configurations. For Experiment 3, the testbed produces, transmits, receives, and writes to disk roughly 1.6 billion packets carrying nearly 1.5 TB of network traffic over the course of 6 hours.

Additionally, the monitor system runs a special version of the pktgen traffic generator that transmits specially addressed packets once every second for 275 seconds—most of each of the ~300 second trial durations. These packets become interspersed within the network traffic load provided by the spitter test system. In and of themselves, these marked packets provide no noticeable workload increase to the system; however, the SUT must search for these marked packets in the packet data it has stored. This workload is considerable given each trial can write between 4 GB and 40 GB to disk.

During Experiment 3, the SUT runs a query script that initiates queries for packet data, starting 60 seconds into the trial until the end of the trial. This script uses TCPDUMP to look through all captured data for the marked packets and to copy the packets to a temporary query response file. The script then reports the number of marked packets found in the captured data and the number of seconds required to accomplish the query-response cycle. Once one cycle is complete, the script will wait one second and run again. The task becomes more difficult for the SUT as the trial progresses, since more data is available to search through.

The query workload impact on the system is multifaceted, in that the SUT CPU resources are being consumed by an additional task, and the storage device can no longer be dedicated to writing alone. The disk must satisfy both the

write and read demands, significantly impacting both the write performance and dropped packet rate. It is for these reasons, that additional lower-end bitrate levels were included in this phase of experiment. The metrics of interest for the query-response cycle are: the query-response cycle delay in seconds, the number of queries successfully completed, and the percentage of the marked packets found. Once the workload begins to negatively impact the dropped packet rate, the reliability of the query finding all marked packets is questionable due to the rationale that some of the marked packets may be among those dropped. Over sufficient quantities of time, the packet drop rate and missed mark rate should converge to the same percentage.

4.9.5 Methodology Summary. This chapter discusses the methodology used to evaluate the performance of the N^{2}d capture system under various workloads and two CUT options. Experiments 1 and 2 evaluate key internal characteristics of the SUT performance by isolating the workloads and system factors. Experiment 3 evaluates the key external characteristics of the SUT by applying the query workload during the test. All experiments evaluate the impact of CUT selection to the overall SUT performance. Statistical tests are then used to analyze the comparative effectiveness of each CUT option to provide N^{2}d services under a various workloads.

V. Analysis

THIS chapter presents and analyzes the results of the experiments as described in Chapter IV. Sections 5.1 through 5.3 discuss the results for the system performance metrics and behavior of the system in light of the metric data. Finally, an overall analysis of the experimental data is presented in Section 5.4.

5.1 Results and Analysis of Experiment 1

Experiment 1 gathers system metrics for each CUT option of the SUT, while storing the packet capture data to a real storage device. This experiment measures performance data for the capture and storage process alone; no data queries occurred during this phase of testing. The metrics for Experiment 1 are categorized by trial set. There are 12 trial sets, each representing 5 independent trials for various combinations of bitrate and packet size. Rather than exploring the metrics for each trial set independently, Section 5.1.1 presents an overview of the key results for the experiment, and the following sections detail characteristics of interest.

5.1.1 Experiment 1 Overview. The three graphs in Figure 5.1 summarize the key metrics for all trial sets where packet sizes are randomized. Of the three options for the packet size factor, randomized sizing is the most representative of standard Internet traffic. Trial sets using minimum packet size (64 Bytes) provide insight into the behavior of the system with a worst-case overhead to data delivery ratio, and trial sets with maximum packet size (1500 Bytes) provide insight into the behavior of the system with a best-case overhead to data delivery ratio.

Inclusion of the results for trial sets with minimum sized packets, introduces too much confusion to accurately represent the behavior of the SUT. Figure 5.2 shows the impact of packet size to the CPU utilization and packet

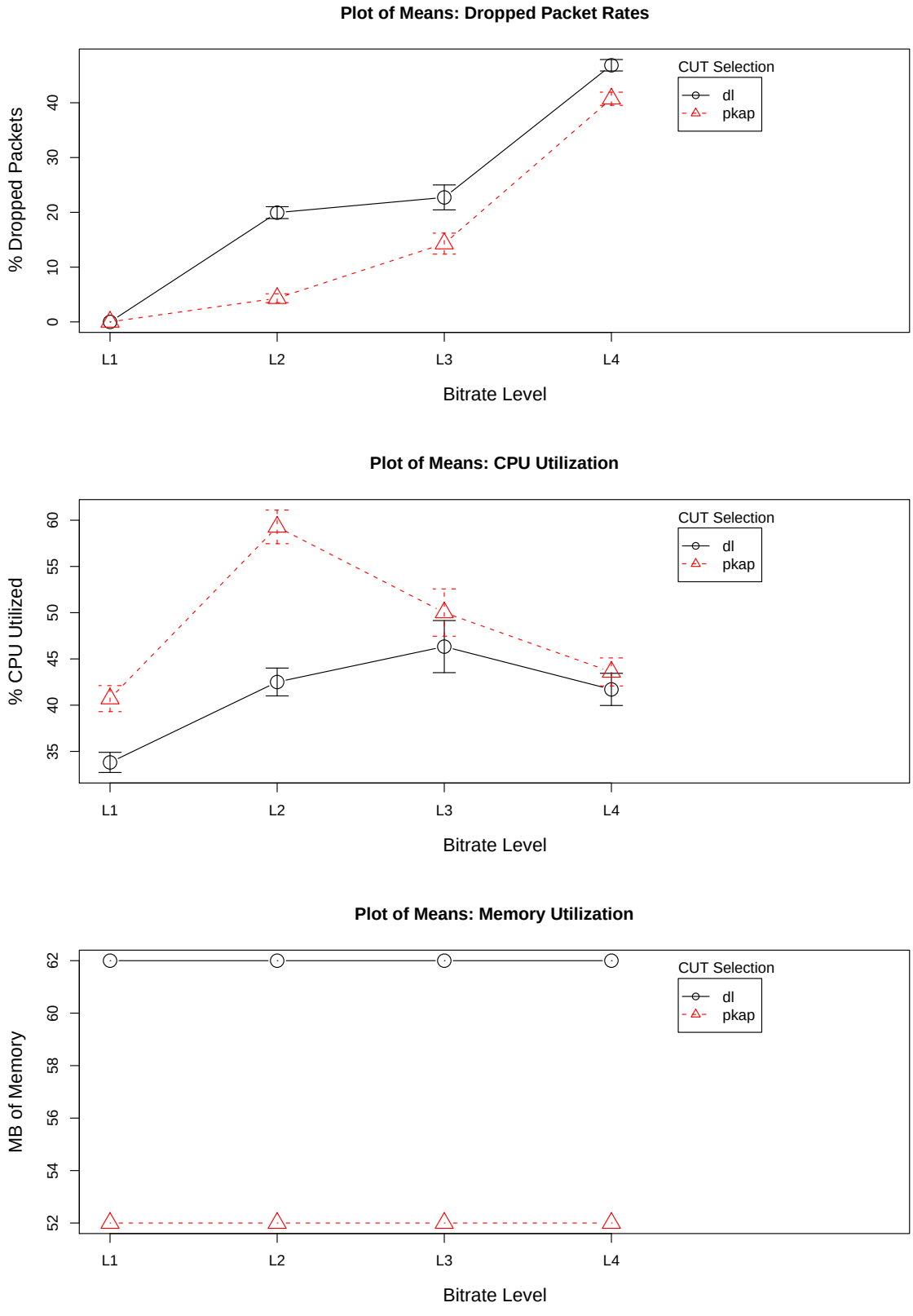


Figure 5.1: Experiment 1: Summary Graphs of Key Metrics: Whiskers represent a 95% confidence interval

drop rate. The whiskers of the graphs show the range for a confidence interval of 95%. At bitrate level 3, a statistical difference does exist between trials with a random size distribution and trials with maximum sized packets; however, the following traits are clearly seen:

1. Trials with maximum size and random size packets behave similarly in packet loss and CPU characteristics
2. Trials with minimum packet size differ noticeably in both packet loss and CPU characteristics

5.1.2 Packet Drop Rate. Outside of level 1 bitrates, where both CUT options are fully capable of zero packet loss, the PKAP kernel module packet drop rate is lower than the DaemonLogger user-space application. Table 5.1 displays the results of a two-variable t-test performed on the packet drop rate for each of the 4 distinct bitrate levels, factored by CUT selection.

$$H_0: p(PktDropRate_{PKAP}) = p(PktDropRate_{DL})$$
$$H_0: p(PktDropRate_{PKAP}) < p(PktDropRate_{DL})$$

The table indicates that the null hypothesis should be rejected for every bitrate level that includes dropped packets. The PKAP kernel module is capable of capturing more packets to disk than the DaemonLogger user-space application. Using additional data collected during the experiment, Figure 5.3 graphically displays the difference between the number of KiloBytes per second (KBps) written to disk by each CUT during the experiment. Due to the multiple dependencies of an N^{2}d capture system, the actual data write rate varies as the network load changes; however, the data shows on average that the kernel-space implementation is capable of writing 15-20% more KBps disk than the user-space application, with a 95% confidence interval.

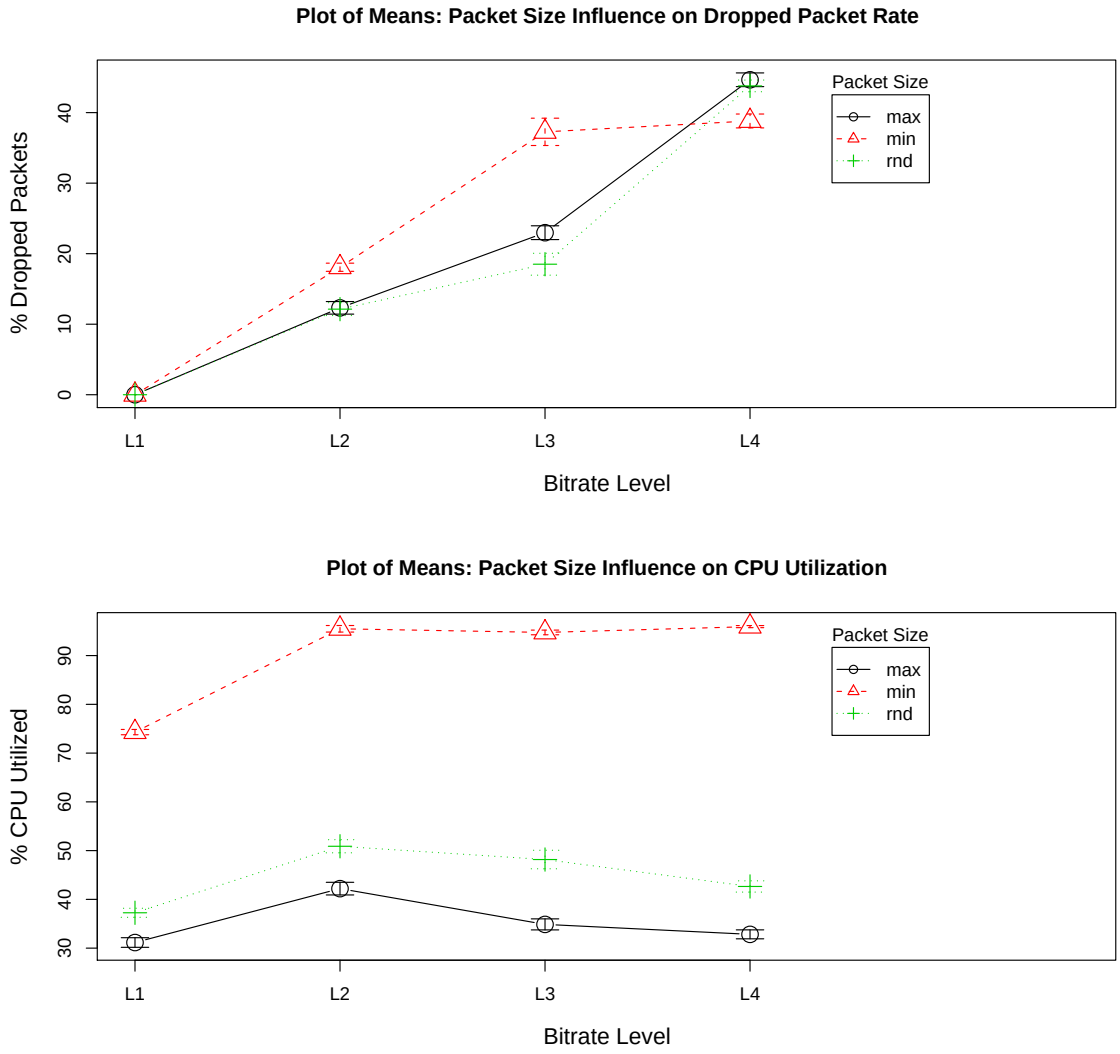


Figure 5.2: Experiment 1: Packet Size Impact: Depicts the impact of packet size in both CPU utilization and Dropped Packet Rate

Table 5.1: Experiment 1 Hypothesis Testing of Dropped Packet Rate

Alternative Hypothesis (95% Confidence Interval)	Scope	estimate	t-ratio	df	p-value
$p(PktDrpRate_{PKAP}) < p(PktDrpRate_{DL})$	Exp 1	0.088552	19.0161	7473.161	4.45e-79
$p(PktDrpRate_{PKAP}) < p(PktDrpRate_{DL})$	Lvl 1	0	N/A	N/A	N/A
$p(PktDrpRate_{PKAP}) < p(PktDrpRate_{DL})$	Lvl 2	0.156176	22.65842	589.7446	1.53e-82
$p(PktDrpRate_{PKAP}) < p(PktDrpRate_{DL})$	Lvl 3	0.084359	5.605101	301.2780	2.35e-8
$p(PktDrpRate_{PKAP}) < p(PktDrpRate_{DL})$	Lvl 4	0.061188	7.545896	625.5938	7.96e-14

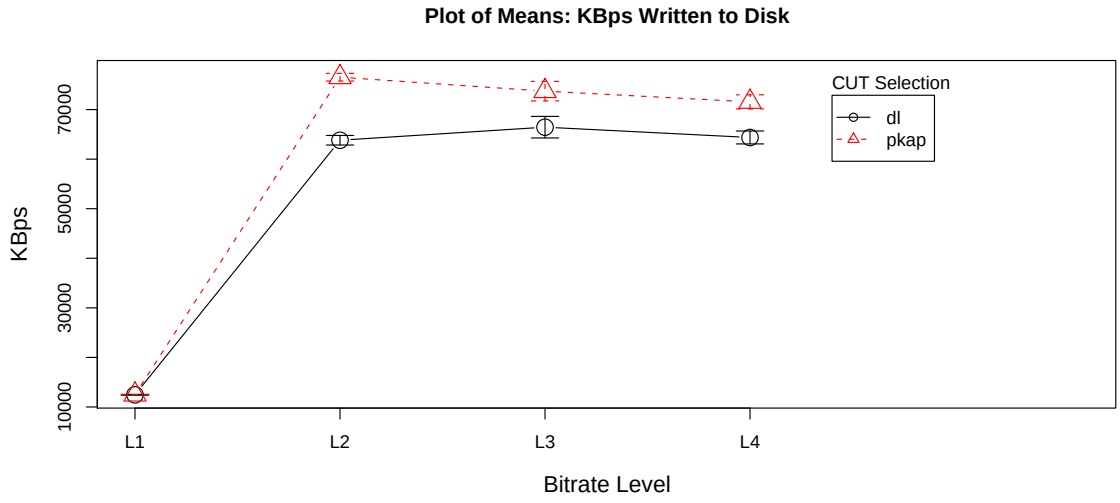


Figure 5.3: Experiment 1: KBps Written / Bitrate Level: Whiskers represent a 95% confidence interval

5.1.3 CPU Utilization. As Figure 5.1 summarizes in the CPU Utilization graph, the original alternate hypothesis that a kernel-mode Ndd application would reduce CPU utilization appears to be shown false.

$$H_{0:}p(CPU_{pkap}) = p(CPU_{DL})$$
$$H_{A:}p(CPU_{pkap}) < p(CPU_{DL})$$

Table 5.2 shows the results of a two-variable t-test performed on CPU utilization and CUT selection for each of the 4 distinct bitrate levels. The tables indicate that the inverse of the original alternate hypothesis is valid for all cases but bitrate level 3, where there is insufficient evidence to reject the null hypothesis. Though this metric alone indicates a failure of the PKAP kernel module to achieve one of its objectives (a reduction in CPU Utilization), Section 5.1.5 analyzes the relationship between metrics and provides insight into the ultimate effect on the SUT.

5.1.4 Memory Utilization. The memory utilization graph in Figure 5.1 depicts the measurable memory utilization throughout all experiments. PKAP utilizes 52 MB of memory, and DaemonLogger utilizes 62 MB of memory; this does not vary with packet size, bitrate, or any other runtime variable. The use of the PF_RING ring buffer, sized upon initialization, makes up the majority of the memory utilized by either CUT option. Both CUT options used PF_RINGs that were identically initialized, and they both experience the same changes in memory usage when the snaplen or the number of ring slots is changed. Outside ring memory consumption, PKAP saves memory by not buffering writes to disk and not copying the data from the ring to use it.

Section 3.1.1 states that one objective for the PKAP proof of concept is to reduce memory utilization, and Section 4.5 lists memory utilization as a system metric. The objective drove design decisions to reduce memory footprint. The memory utilization of PKAP is consistently 16% less than of DaemonLogger,

Table 5.2: Experiment 1 Hypothesis Testing of CPU Utilization: The first table provides clear evidence that the null hypothesis cannot be rejected, while the second table shows that the inverse alternate hypothesis is valid for all but bitrate level 3, with a 95% confidence interval

Original Alternative Hypothesis (95% Confidence Interval)	Scope	estimate	t-ratio	df	p-value
$p(CPU_{PKAP}) < p(CPU_{DL})$	Exp 1	-7.8496	-11.8092	2179.89	1
$p(CPU_{PKAP}) < p(CPU_{DL})$	Lvl 1	-6.8988	-7.6231	582.1569	1
$p(CPU_{PKAP}) < p(CPU_{DL})$	Lvl 2	-16.7715	-13.9820	614.6536	1
$p(CPU_{PKAP}) < p(CPU_{DL})$	Lvl 3	-3.6823	-1.9120	307.1885	0.971596
$p(CPU_{PKAP}) < p(CPU_{DL})$	Lvl 4	-1.8878	-1.6100	626.1343	0.946047

Inverse Alternative Hypothesis (95% Confidence Interval)	Scope	estimate	t-ratio	df	p-value
$p(CPU_{PKAP}) > p(CPU_{DL})$	Exp 1	-7.8496	-11.8092	2179.89	1.53e-31
$p(CPU_{PKAP}) > p(CPU_{DL})$	Lvl 1	-6.8988	-7.6231	582.1569	5.06e-14
$p(CPU_{PKAP}) > p(CPU_{DL})$	Lvl 2	-16.7715	-13.9820	614.6536	4.51e-39
$p(CPU_{PKAP}) > p(CPU_{DL})$	Lvl 3	-3.6823	-1.9120	307.1885	0.02840
$p(CPU_{PKAP}) > p(CPU_{DL})$	Lvl 4	-1.8878	-1.6100	626.1343	0.05395

so the metrics indicate that the objective to reduce memory usage has been achieved.

5.1.5 Summary Analysis. Individual system metrics indicate that the PKAP kernel thread drops fewer packets, consumes more CPU resources, and uses less memory than the DaemonLogger user-space application. While the indications provided by the metrics discretely do provide some information about the impact of moving the N^2d capability to kernel-space, it does not provide the full picture without probing the dependencies between the metric data.

Of particular note is the relationship between dropped packets and CPU utilization. Figure 5.4 provides a typical interaction between the two metrics. The graph's data set is from Experiment 1, bitrate level 2. Though only a single trial run, the relationship between the two metrics is representative of all trial runs that resulted in dropped packets. Any increase in the dropped packet rate reduces the CPU utilization of the CUT. As the PKAP kernel module succeeds in reducing the dropped packet rate, it fails to reduce the CPU utilization. This relationship exists since CUT processing requirements only occur for packets that it retrieves from the PF_RING ring buffer. Dropped packets never reach the CUT—thereby reducing the ultimate workload of the CUT at any given network bitrate. The effect of the dropped packets is similar to the effect that filtering would provide to the CUT, though the dropped packets would be of intentional choosing. Since the ultimate internal priority is capturing the data accurately, the trade-off of CPU utilization for improved capture integrity is still aligned with the aims of the research.

Experiment 1 trials using bitrate level 1 reveal that the PKAP CUT option does use a greater percentage of CPU resources even when the difference cannot be attributed to dropped packets. The data does not indicate

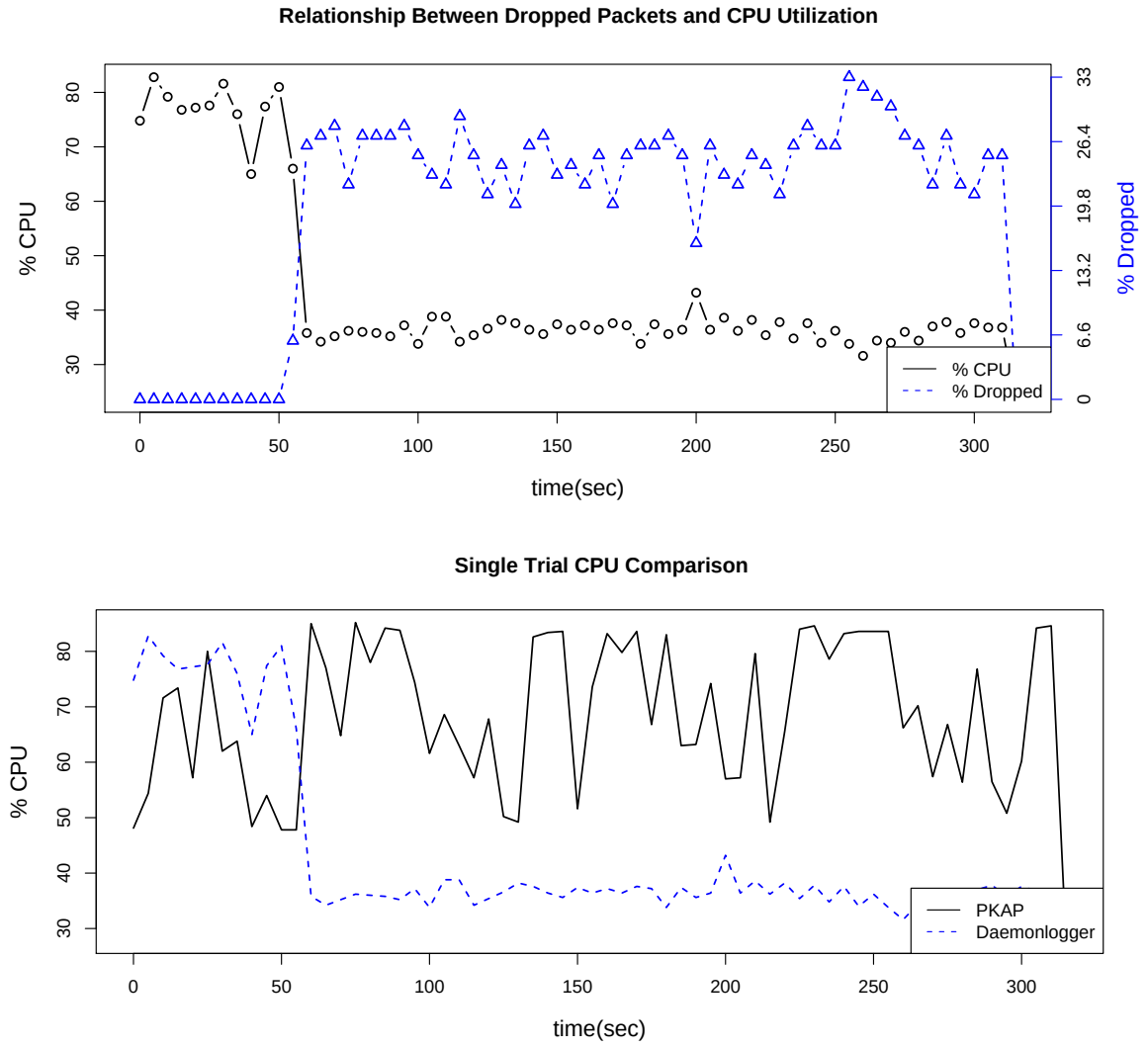


Figure 5.4: Experiment 1: Relationship Between Dropped Packets and CPU Utilization: (Graph A) The scales of the Y-axes differ to optimize visual recognition of the relationship between the affect the dropped packet rate has on the CPU utilization. The graph is composed of a single trial; however, it is representative of the relationship in all trials. (Graph B) Compares the CPU utilization between CUT options for the same trial as graphed in Graph A; PKAP dropped no packets, and the DaemonLogger drop rate is as graphed in Graph A.

5.2 Results and Analysis of Experiment 2

Experiment 2 gathers system metrics for each CUT option of the SUT, while storing the packet capture data to the `/dev/null` device. This experiment measures performance data for the capture and storage process alone; no data queries occurred during this phase of testing. The metrics for Experiment 2 are categorized by trial set. There are 12 trial sets, each representing 5 independent trials for various combinations of bitrate and packet size. Section 5.1.1 presents an overview of the key results for the experiment, and the following sections detail characteristics of interest.

5.2.1 Experiment 2 Overview. The three graphs in Figure 5.5 summarize the key metrics for all trial sets where packet sizes are randomized. While imperfect, this setting is the most representative size distribution of standard Internet traffic. Trial sets using minimum packet size (64 Bytes) provide insight into the behavior of the system with a worst-case overhead to data delivery ratio, and trial sets with maximum packet size (1500 Bytes) provide insight into the behavior of the system with a best-case overhead to data delivery ratio.

Rationale for the general exclusion of trials with minimum or maximum sized packets from the overall summary is similar to the explanation provided during the analysis of Experiment 1. The depiction of the influence of packet size on the SUT performance during Experiment 2 is shown in Figure 5.6.

The use of the data regarding packet size influence is somewhat different for Experiment 2 analysis. Experiment 2 strives to provide some assessment and indication for performance beyond limits of what real storage hardware is available for testing. This introduces specific departure from real-world testing, and thus the trial data including minimum and maximum size packets is useful for revealing certain computational aspects of CUT selection.

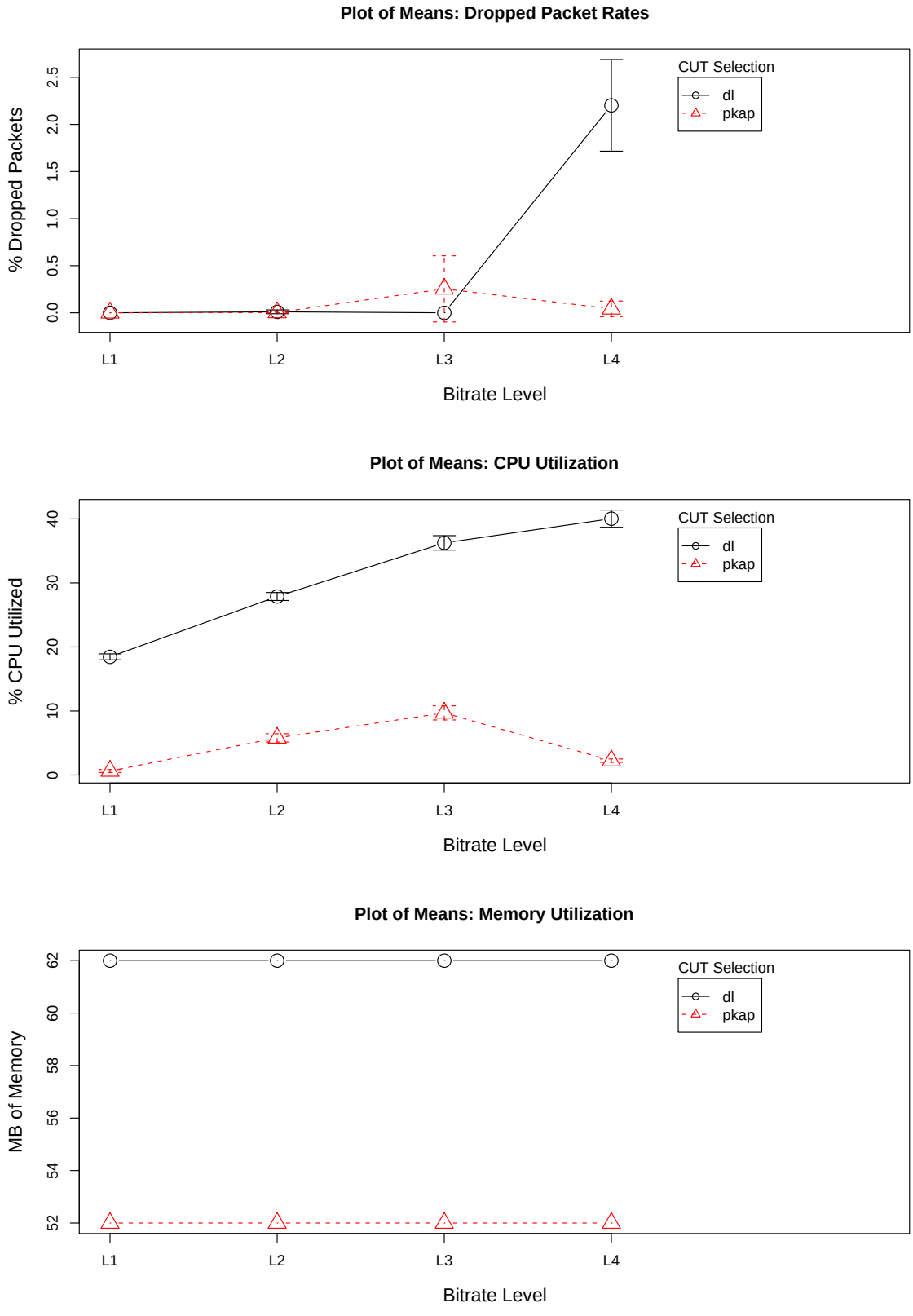


Figure 5.5: Experiment 2: Summary Graphs of Key Metrics

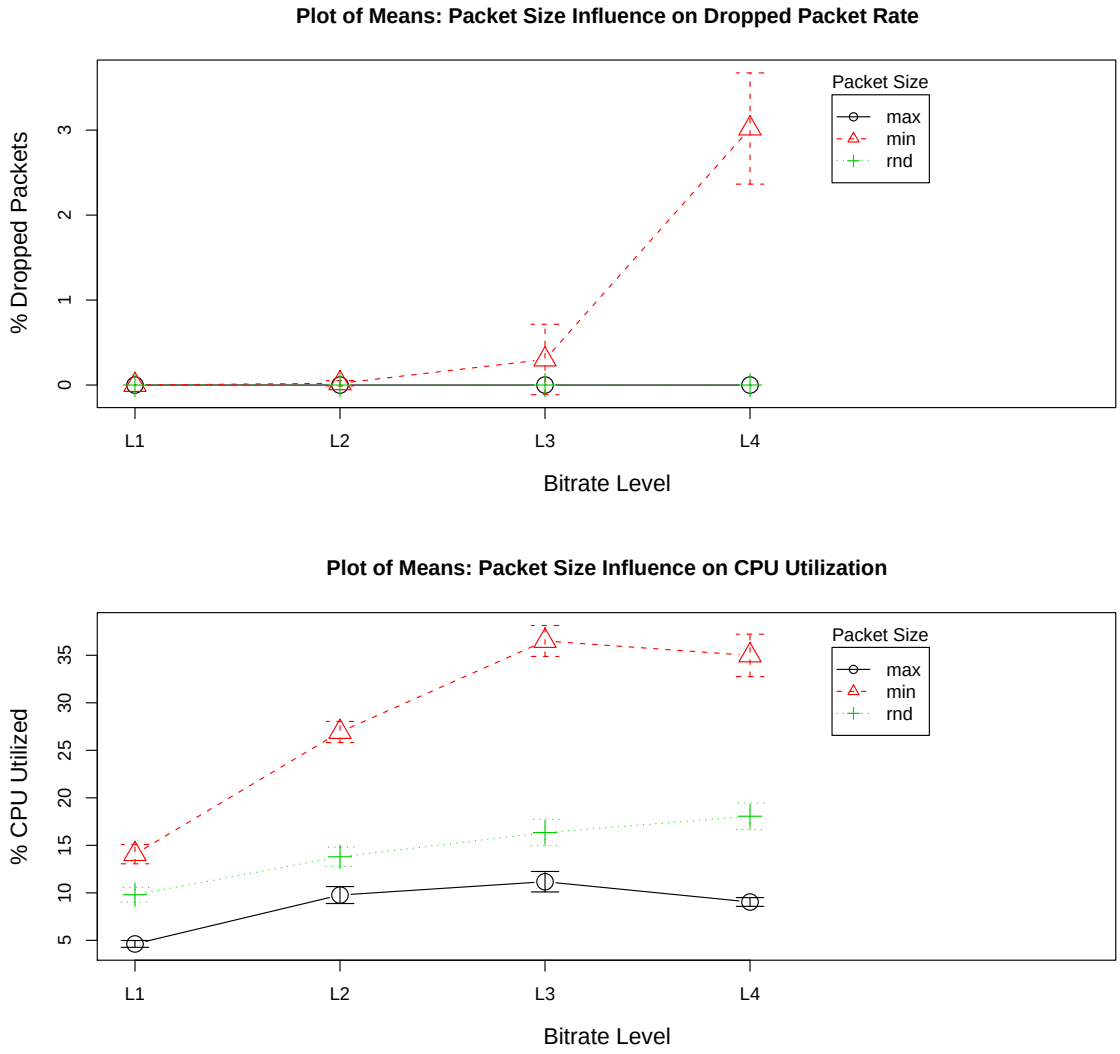


Figure 5.6: Experiment 1: Packet Size Impact: Depicts the impact of packet size in both CPU utilization and Dropped Packet Rate

5.2.2 Packet Drop Rate. Contrary to Experiment 1, where packet drop rate differences emerge beyond level 1 bitrates, the use of the `/dev/null` device for writing captured data maintains very low drop rates until much higher bitrates. This shift is due to the fact that nearly every write is virtually non-blocking. Though all write activity (even those to `/dev/null`) is a potential blocking call, the fact that the filesystem delivers the bytes to the bit bucket as fast as it gets them, removes one of the most significant delays of an Nd system. The bytes never hit the buffer cache; and since the data written to `/dev/null` never produces a dirty page, the bdflush kernel daemon never increases the CPU usage for write tasks. It is for these reasons that the results of Experiment 2 are useful only for identification of processing behavior outside of the storage task.

The results of a two-variable t-test performed on the packet drop rate for each of the 4 distinct bitrate levels, factored by CUT selection, reveal no statistically significant packet loss for any bitrate when considering trials with randomly selected packet sizes or maximum packet sizes; therefore, Table 5.3 displays the results of a two-variable t-test performed on the packet drop rate of trials with minimum size packets for each of the 4 distinct bitrate levels, factored by CUT selection.

$$H_0{:} p(PktDropRate_{PKAP}) = p(PktDropRate_{DL})$$
$$H_0{:} p(PktDropRate_{PKAP}) < p(PktDropRate_{DL})$$

The table indicates that the null hypothesis should be rejected overall and that alternate hypothesis is valid. When the t-test is scoped by level, the null hypothesis cannot be rejected with a 95% confidence interval for trials with bitrate levels 2 and 3. The practical difference between methods is only visible during trials with bitrate level 4, and the difference is estimated to be only 5%. Yet, since bitrate level 4 is 340 Mbps for trials with minimum size packets, the

Table 5.3: Experiment 1 Hypothesis Testing of Dropped Packet Rate

Alternative Hypothesis (95% Confidence Interval)	Scope	estimate	t-ratio	df	p-value
$p(PktDrpRate_{PKAP}) < p(PktDrpRate_{DL})$	Exp 2	0.01508	6.87072	2206.68	4.14e-12
$p(PktDrpRate_{PKAP}) < p(PktDrpRate_{DL})$	Lvl 1	0	N/A	N/A	N/A
$p(PktDrpRate_{PKAP}) < p(PktDrpRate_{DL})$	Lvl 2	0.00022	0.68979	338.0031	0.24540
$p(PktDrpRate_{PKAP}) < p(PktDrpRate_{DL})$	Lvl 3	-0.00599	-1.42361	328	0.92224
$p(PktDrpRate_{PKAP}) < p(PktDrpRate_{DL})$	Lvl 4	0.05350	9.02628	464.768	2.36e-18

end result of a 5% difference translates to nearly 700,000 packets and 45 MB of data per second.

5.2.3 CPU Utilization. As graphically depicted in the CPU Utilization graph from Figure 5.5, there is ample evidence to reject the original null hypothesis in favor of the alternate hypothesis, which states that a kernel-mode Nd application would reduce CPU utilization.

$$H_{0:}p(CPU_{pkap}) = p(CPU_{DL})$$
$$H_{A:}p(CPU_{pkap}) < p(CPU_{DL})$$

Table 5.4 shows the results of a two-variable t-test performed on CUT selection for each of the 4 distinct bitrate levels. The table indicates that the PKAP kernel module clearly utilizes the CPU significantly less than the DaemonLogger user-space application. The data shows savings between 20-35% at a 95% confidence interval.

Since PKAP does not buffer writes, it is not penalized the way DaemonLogger is in this artificial circumstance. DaemonLogger must accomplish to work to copy the ring slots to a stream buffer and perform the expensive copy

Table 5.4: Experiment 2 Hypothesis Testing of CPU Utilization

Original Alternative Hypothesis (95% Confidence Interval)	Scope	estimate	t-ratio	df	p-value
$p(CPU_{PKAP}) < p(CPU_{DL})$	Exp 2	26.02774	109.7004	1765.758	0
$p(CPU_{PKAP}) < p(CPU_{DL})$	Lvl 1	19.57145	447.5581	335.3397	0
$p(CPU_{PKAP}) < p(CPU_{DL})$	Lvl 2	24.56123	72.62268	547.5201	1.58e-283
$p(CPU_{PKAP}) < p(CPU_{DL})$	Lvl 3	22.07348	34.35730	165.9576	1.19e-77
$p(CPU_{PKAP}) < p(CPU_{DL})$	Lvl 4	35.66156	365.6073	455.083	0

from user-space to kernel-space even though the packets are immediately discarded by the filesystem. The reasons that PKAP requires less CPU in this circumstance in and of itself is of no practical value in real-world performance; however, the data does point to a gain in write efficiency from within kernel-space, and it indicates what percentage of the CPU resources are involved in the CUT's handling of the packet from the ring to the filesystem in a non-blocking scenario.

5.2.4 Memory. As mentioned in Section 5.1.4, the memory utilization is static and not depend ant upon system factors or workload levels. The memory usage is not changed, and in the case of storage to `/dev/null`, the additional memory (and associated copies) used by the DaemonLogger user-space application consumes both memory and CPU resources without benefit.

5.2.5 Summary Analysis. Given the artificial nature of Experiment 2, the resultant data and analysis not directly applicable to a real-world N^2 capability; however, both the data and charts seem to indicate that a kernel-space capture capability can derive more benefit from faster storage than a user-space capture capability. The built in buffering from the user-space capture capabil-

ity can be helpful in accommodating comparably slower storage devices; yet, if storage speeds increase, the extra cost of copying memory additional times could prevent full utilization of the storage device performance.

5.3 Results and Analysis of Experiment 3

Experiment 3 gathers system metrics for each CUT option of the SUT, while storing the packet capture data to a real storage device. This experiment measures performance data for the capture, storage, and data query processes. The metrics for Experiment 3 are categorized by trial set. There are 7 trial sets, each representing 5 independent trials for 7 levels of bitrate. Since this scenario most closely represents the real-world workload of an N^2_d system, the packet sizes are configured to be randomly selected for all sets. Figure 5.7 presents an overview of the key process-based results for the experiment, while Figure 5.8 presents an overview of the key query-based results for the experiment.

5.3.1 Experiment 3 Overview. The graphs in Figure 5.7 and Figure 5.8 summarize the key metrics for all trial sets in Experiment 3. All trial sets in this experiment use randomly sized packets, ranging from 64 Bytes to 1500 Bytes. During trials of all bitrate levels, a query process on the SUT repeatedly searches through all packets recorded during the trial for the marked packets of interest.

Since Experiments 1 and 2 cover the first three system metrics under exclusive network workload, the following sections detail the remaining query metrics and the affect of the query metrics on the first three system metrics.

5.3.2 Query Metrics. The query process on the SUT records the number of queries successfully completed, the time delay for response, and the number of packets found. The network and query workloads combine to effectively

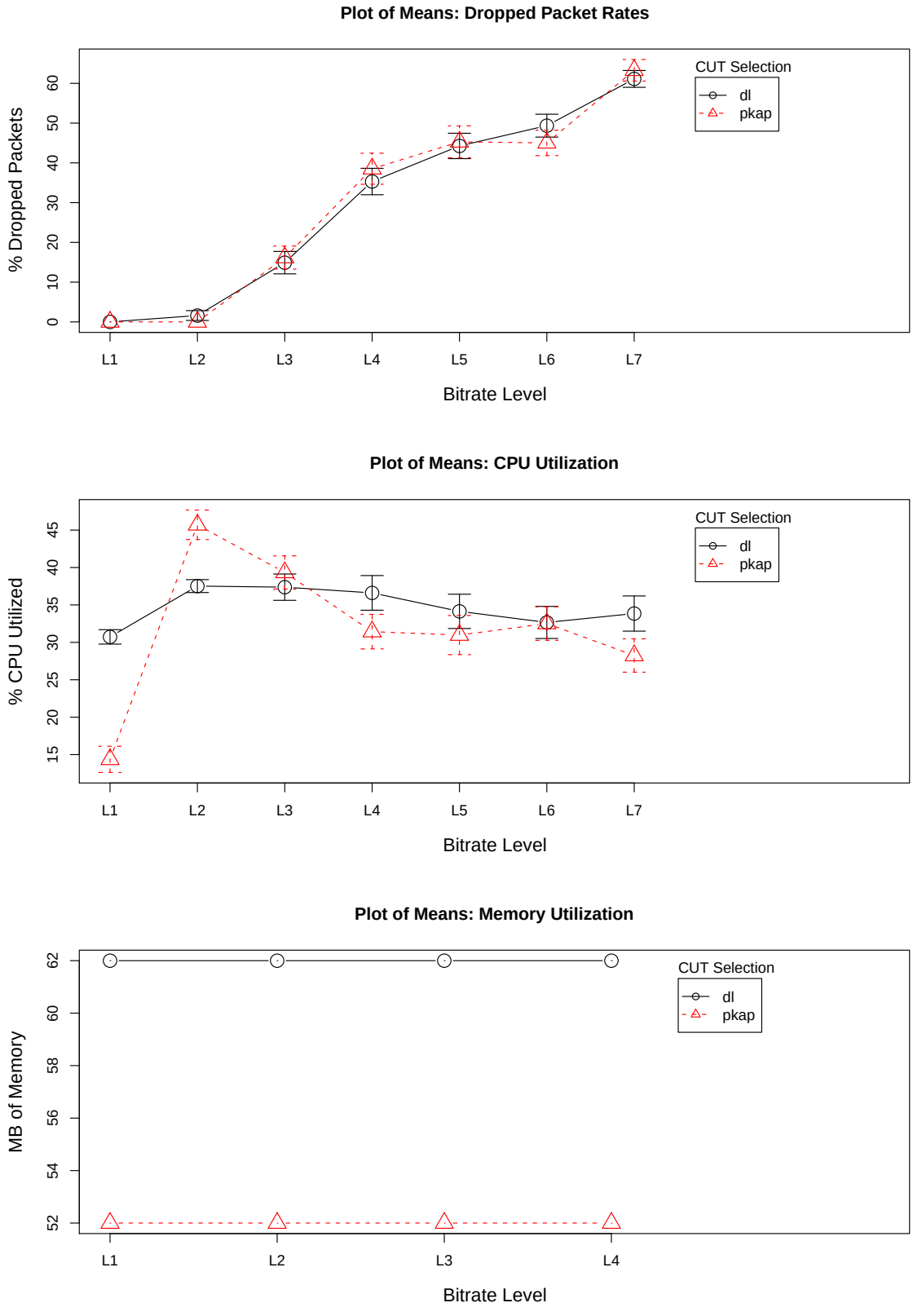


Figure 5.7: Experiment 3: Summary Graphs of Key Process-Based Metrics

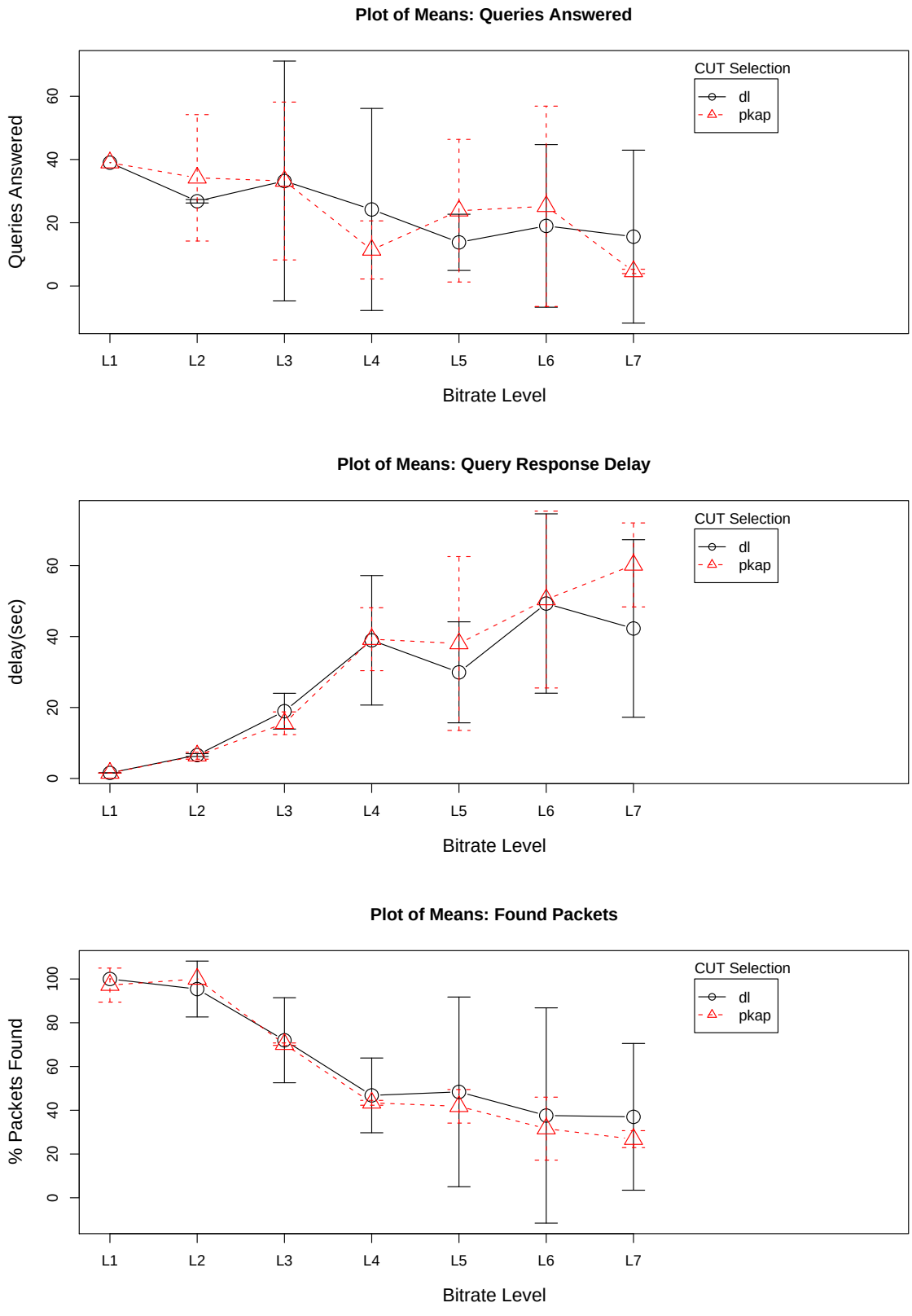


Figure 5.8: Experiment 3: Summary Graphs of Key Query-Based Metrics

exercise and assess the SUT. Figure 5.8 depicts an overview of these metrics. High-level observations from the graphs indicate the following:

- The mean values for the number of queries answered and the delay of the responses have wide ranges for 95% confidence intervals
- The erratic patterns for the number of queries answered and the delay of the responses
- As the bitrate level increases:
  - The number of queries answered decreases
  - The delay for the query responses increases
  - The number of packets found decreases
- Alone, the plots of means for the query metrics do not indicate a significant difference in performance between CUT selections

The goal of this research with respect to queries for captured data is that a kernel-space Nd[2] capability would retain user-space disk responsiveness to satisfy live queries for the captured data. Therefore, satisfaction of this goal is achieved by a lack of evidence to reject the null hypothesis, as such:

$$H_{0:} p(QryCount_{PKAP}) = p(QryCount_{DL})$$
$$H_{A:} p(QryCount_{PKAP}) < p(QryCount_{DL})$$

$$H_{0:} p(QryDelay_{PKAP}) = p(QryDelay_{DL})$$
$$H_{A:} p(QryDelay_{PKAP}) > p(QryDelay_{DL})$$

$$H_{0:} p(QryFound_{PKAP}) = p(QryFound_{DL})$$
$$H_{A:} p(QryFound_{PKAP}) < p(QryFound_{DL})$$

Table 5.5: Experiment 3 Hypothesis Testing of Query Performance

Alternative Hypothesis (95% Confidence Interval)	Scope	estimate	t-ratio	df	p-value
$p(QryCount_{PKAP}) < p(QryCount_{DL})$	Exp 3	0.51429	0.11962	67.3228	0.45257
$p(QryDelay_{PKAP}) > p(QryDelay_{DL})$	Exp 3	-2.00229	-0.42857	65.1373	0.33483
$p(QryFound_{PKAP}) < p(QryFound_{DL})$	Exp 3	0.00085	0.01065	65.6037	0.49577

Table 5.5 shows the results for a two-variable t-test performed on the above metrics and bitrate, factored by CUT selection. In all cases, there is insufficient evidence that the null hypothesis can be rejected. The data indicates that the objective to retain the query performance of a user-space CUT when using the kernel-space implementation is achieved.

5.3.3 Packet Drop Rate. The packet drop rate not only describes the performance of the storage system; it also contributes directly to the successful search for packets of interest—marked packets, in the case of Experiment 3. The relationship between the packet drop rate and CPU utilization largely mimics the behavior seen in Experiment 1. As more packets are dropped, the CPU utilization decreases.

Table 5.6 shows the results for a two-variable t-test on packet drop rate and bitrate level, factored by CUT selection. There is insufficient evidence to reject the null hypothesis, indicating that CUT selection does not significantly impact packet drop rates when a query workload is present. While the dropped packet rate of DaemonLogger under a query workload is estimated to be slightly less than 0.1% better than PKAP, this gap is consumed by the 2.6% range of the 95% confidence interval.

Since the null hypothesis is rejected in Experiment 1 and accepted in Experiment 3, this identifies that the query workload changes the impact of CUT

Table 5.6: Experiment 3 Hypothesis Testing of Dropped Packet Rate

Alternative Hypothesis (95% Confidence Interval)	Scope	estimate	t-ratio	df	p-value
$p(PktDrpRate_{PKAP}) < p(PktDrpRate_{DL})$	Exp 3	-0.00963	-0.94478	4105.14	0.82759

Table 5.7: Experiment 3 Hypothesis Testing of CPU Utilization

Original Alternative Hypothesis (95% Confidence Interval)	Scope	estimate	t-ratio	df	p-value
$p(CPU_{PKAP}) < p(CPU_{DL})$	Exp 3	3.39744	5.43562	3942.623	2.90e-08

selection on the packet drop rate. A potential reason for the comparatively improved performance of the user-space CUT is the write buffering used by the DaemonLogger application. Since the PKAP module uses the slot memory directly for writing to the filesystem, any blocking that occurs during writes can quickly consume the free ring slots. This effect can be reduced somewhat by DaemonLogger's buffering, as the ring is emptied quickly by a simple `memcpy()`—providing another buffer to reduce the cost of storage device contention.

5.3.4 CPU Utilization. Table 5.7 shows the results of a two-variable t-test performed on CPU utilization and CUT selection during Experiment 3. There is sufficient evidence to reject the null hypothesis in favor of the alternate hypothesis, indicating that the kernel-space CUT reduces CPU utilization when a query workload is present. The estimated reduction of the kernel-space CUT is 3%, at a 95% confidence interval. In practice, the difference of 3% utilization when maximum utilization is under 50% is insignificant; however, this behavior is a reversal of that found during Experiment 1 tests. This could indicate that non-buffered, kernel-space file writing incurs reduced contention in a non-write-exclusive scenario.

5.3.5 Memory. As mentioned in Section 5.1.4, the memory utilization is static and not depend ant upon system factors or workload levels. Based on the indications of the buffering benefits mentioned in Section 5.3.3, the memory DaemonLogger uses in stream buffering writes appears to be beneficial in during times of heavy disk I/O storms. Alternatively, the buffering can be increased for either CUT option by initializing the PF_RING ring buffer with more slots.

5.3.6 Summary Analysis. The query workload imposed during Experiment 3 certainly impacted the comparative performance of the CUT selections versus that described during Experiment 1 analysis. In total, the PKAP packet drop rate is indistinguishable that of DaemonLogger, while demanding slightly less CPU utilization and maintaining query performance.

5.4 Overall Analysis

To recap, the key aspects of the PKAP kernel module during the three experiments:

- The PKAP kernel module improves upon user-space Nd^{2}d capture performance either by:
  - Capturing a greater percentage of packets, with increased CPU utilization, when the storage writes have exclusive access to the storage device
  - Capturing the same percentage of packets, with reduced CPU utilization, when the storage writes face heavy disk I/O contention
- The PKAP kernel module uses less memory; however, the practical difference of the reduction is insignificant on today's computing systems

- The PKAP kernel module does not negatively alter the query response performance characteristics of the system compared to a user-space CUT selection

VI. Discussion

THIS chapter discusses the overall conclusions drawn during the course of the research. Section 6.1 addresses each research objective and determines whether or not the objectives were achieved, and Section 6.2 presents the significance of the research. Finally, Section 6.3 recommends possible future directions for related research.

6.1 Conclusions

6.1.1 Construct a Kernel-space N^2d Capability. The first goal of this research is to design and build a proof of concept N^2d capture capability that resides within kernel-space. The basic functions of an N^2d capture capability are the ability to promiscuously sniff packets, filter out packets that are not of interest, and store the packets to a permanent storage device in a format that can be used to reconstruct the network activity after the fact. Additionally, the system must be capable of long-term, unattended fielding, which requires that system to police its own storage system by rolling old log information as space for new data is required.

To accomplish this goal, the PF_RING kernel module is modified to export a given ring so the memory can be addressed by another kernel module and to provide necessary functions to select the ring to be shared. With the modified PF_RING kernel module in place, the PKAP Linux kernel module is designed and developed to create a PF_RING ring-buffer, initialize it, convert it to a kernel-space ring, and consume the ring's memory slots in the same way that the pfring library functions in user-space. Extending the role of a typical Linux kernel module beyond its traditional role, the PKAP module is able to write the captured packets directly to a pcap-formatted log file and orchestrate the log rotation for long-term, unattended deployment.

The PKAP kernel module is loaded onto a Linux test system and tested against real network traffic. Demonstrating the ability to capture packets, fil-

ter them, store them, and maintain their lifecycle through live network testing accomplishes the first research goal.

6.1.2 Reduce Packet Drop Rate of the Capture Process. This second research goal is to improve the performance characteristics of an N^{2}d capture system by employing the kernel-space capture capability in place of a user-space implementation. The specific metric for improvement in this case is the reduction of the packet drop rate during capture of network traffic.

Testing the system with live network traffic reveals that PKAP reduces the packet drop rate by an average of 8.9% (using a 95% confidence interval) when disk I/O contention is low. In cases where the disk I/O contention is high, the PKAP module does not reduce the drop rate from that of a user-space application; however, during this scenario, the PKAP module produces a favorable reduction in CPU utilization compared to the user-space implementation. Given the practical use of an N^{2}d system to capture network data for the majority of its runtime and be required to satisfy queries for data periodically, the changes that are made by PKAP to the packet drop rate, and its depend ant metrics, meet the second research goal.

6.1.3 Reduce CPU Utilization of the Capture Process. The third research goal is to reduce the resource demands of the N^{2}d capture process by reducing the CPU utilization. Reduction of the CPU utilization is thought possible due to the removal of multiple memory copies and the transition from user-space to kernel-space for writing the packets to disk; however, the reduced CPU utilization is only observable during times of high disk I/O contention. At these times, PKAP reduces the CPU utilization by an average of 3%, at a 95% confidence interval.

When disk I/O contention is low, the PKAP module actually increases CPU utilization of the capture system because of the increased capacity to cap-

ture packets without dropping them. As discussed in Chapter V, the CPU utilization is depend ant on many things, one of which is the packet drop rate. By reducing the packet drop rate by 8.9% on average, the capture system must expend resources to handle the additional workload. During test scenarios of this type, the PKAP module increases CPU utilization by an average of 7.8%, at a 95% confidence interval. The increased CPU utilization is an acceptable trade-off for the accurate capture of a higher percentage of network traffic. This is true particularly since the PKAP module reduces the packet drop rate more than it increases the CPU utilization.

Considering the behavior and benefits of the PKAP module on CPU utilization for situations of both high and low disk I/O contention in light of the priority impact of the packet drop rate on the effectiveness of an N^{2}D capture system, the research goal is achieved.

6.1.4 Reduce Memory Utilization of the Capture Process. The fourth research goal is to reduce the resource demands of the N^{2}D capture process by reducing memory utilization. Reduction of memory utilization is thought possible due to the PKAP module's direct use of the ring memory slot in the call to write the data to disk. The kernel-space implementation uses no stream buffering, since filesystem writes from within the kernel appear to be cheaper than the additional memory copies required to separate the ring consumption from storage tasks or to buffer the data from the ring for fewer large write actions to disk. Through the lack of these copies, the PKAP kernel-space implementation reduces the memory utilization of the N^{2}D task by 10 MB, which under the tested PF_RING configuration equates to a 16% reduction, with a 100% confidence interval. The static reduction of memory utilization achieves this research goal.

6.1.5 Retain Query Response Performance of the Capture Process. The final research goal is to retain the query response performance of the capture system for a given query workload. While other research goals are internally focused, this goal targets the ability of the system to perform the external service—the service which defines the reason for the system's existence.

The query response performance is a tri-faceted metric, encompassing the number of queries during a period of time, the average delay time for each query, and the percent of the known packets of interest found and retrieved by the query. Throughout the trial sets of Experiment 3, the PKAP kernel module does not affect the query response metrics in a statistically distinguishable way from the user-space implementation. The PKAP kernel-space storage of captured packets does not negatively impact the query response performance of the N^2d system, thereby achieving the final research goal.

6.2 Significance

This research provides Law Enforcement, Homeland Defense, the Department of Defense, and industry at large with an improved method for efficiently capturing network traffic to permanent storage. The N^2d capture capability benefits the overall security of information and resources of high value in environments with persistent threats. Any improvement to the efficiency of collection reduces the footprint and increase the effectiveness of the capture capability. The PKAP proof of concept improves the efficiency of the N^2d process, and indications are that it scales better than user-space implementations with increased performance of storage devices.

At the technical level, the PKAP Linux kernel module introduces a stable, high-performance capture application that can both write to disk and orchestrate the rolling log files necessary for long-term deployments of an N^2d service. This research provides a unique way to improve a capture process where the principle goal is storage of the packet streams to disk, and it paves the way for

future research to begin addressing the performance issues of capture systems beyond just the capture side of the equation.

Though PKAP only improves capture performance by reducing the dropped packet rate 9%, that small percentage could translate to losses that approach 90 Mbps on an Gigabit Ethernet link. A common method to obfuscate malicious activity is to surround it by an overwhelming amount of harmless data. If the flood of network traffic increases the probability that key packets are lost, then the evidence gathered decreases in value—the critical attack may even be missed altogether. In light of the value of the data lost, even an improvement of 9% can be significant.

6.3 Future Efforts

The next logical steps for subsequent research include the following list of related efforts:

- Scale the PKAP proof of concept beyond a single Gigabit Ethernet capture interface
- Expand the PF_RING and PKAP kernel modules to capture, filter, and store IPv6 packets
- Evaluate performance with high-end storage devices
- Take further advantage of the kernel-space location through the effective use of the already-parsed `sk_buff` data to build indexes for the stored packets to improve the query response performance through reduction of disk I/O contention
- Improve the run-time control of the PKAP kernel module to provide autonomous survivability before the capture system reaches a failure mode

```
Index: daemonlogger-1.2.1/daemonlogger.c
===================================================================
--- daemonlogger-1.2.1/daemonlogger.c   (revision 88)
+++ daemonlogger-1.2.1/daemonlogger.c   (revision 89)
@@ -216,4 +216,8 @@
 static int maxpct;
 static int prune_flag;
+/*
+ * Option for PKAP comparison
+ */
+static int logtonull = 0;

 static char *interface;
@@ -415,4 +419,11 @@
 {
     time_t currtime;
+    // For PKAP Comparison
+    char*  nullfile = "/dev/null";
+
+    if (logtonull == 1) {
+           return nullfile;
+    }
+    // End of added code for PKAP comparison

     memset(logdir, 0, STDBUF);
@@ -1137,4 +1148,5 @@
     printf("        -u <user name>  Set user ID to <user name>\n");
     printf("        -v              Show daemonlogger version\n");
+    printf("        -X              Log to /dev/null for PKAP \n");
 }

@@ -1153,5 +1165,5 @@

     while((ch = getopt(argc, argv,
-            "c:df:Fg:hi:l:m:M:n:o:p:P:rR:s:S:t:T:u:vz"))!=-1)
```

```
+                "c:df:Fg:hi:l:m:M:n:o:p:P:rR:s:S:t:T:u:vXz"))!=-1)
      {
          switch(ch)
@@ -1300,4 +1312,7 @@
                    prune_flag = PRUNE_OLDEST_IN_RUN;
                    break;
+             case 'X': /* Added for PKAP Comparison */
+                 logtonull = 1;
+                 break;
              default:
                  break;
```

```
Index: pf_ring/linux/pf_ring.h
===================================================================
--- pf_ring/linux/pf_ring.h     (revision 12)
+++ pf_ring/linux/pf_ring.h     (revision 15)
@@ -44,4 +44,8 @@
 #define SO_SET_APPL_NAME                110
 #define SO_SET_PACKET_DIRECTION         111
+#define SO_SET_PKAP_RING                119 /* Export pointer to
+                                               ring so PKAP
+                                               can use it from
+                                               kernel space */

 /* Get */
@@ -454,4 +458,7 @@
    /* Function pointer */
    do_handle_filtering_hash_bucket handle_hash_rule;
+
+  /* PKAP Ring Switch */
+  u_int8_t pkap_ring; // 0=not_pkap 1=pkap
 };

Index: pf_ring/pf_ring.c
===================================================================
--- pf_ring/pf_ring.c   (revision 12)
+++ pf_ring/pf_ring.c   (revision 77)
@@ -74,5 +74,28 @@
 #include <linux/pf_ring.h>

-/* #define RING_DEBUG */
+//#define RING_DEBUG
+
+//#define PKAP_DEBUG
+#ifdef PKAP_DEBUG
+#define pkap_debug(a, args...) do { \
+    printk(KERN_INFO "[PKAP_PF_RING Debug]: " a, ## args) ; \
```

```
+} while(0)
+#else /* DEBUG */
+#define pkap_debug(a, args...)
+#endif /* DEBUG */
+
+#define PKAP_ENABLED
+
+#ifdef PKAP_ENABLED
+struct ring_opt *pkap_pfr;
+EXPORT_SYMBOL(pkap_pfr);
+#endif
+
+
+#ifdef PKAP_DEBUG
+int pkap_test = 777;
+EXPORT_SYMBOL(pkap_test);
+#endif
+
 /* ************************************************ */
 #define TH_FIN_MULTIPLIER      0x01
@@ -187,7 +210,8 @@
 /* Defaults */
 static unsigned int num_slots = 4096;
-static unsigned int enable_tx_capture = 1;
+static unsigned int enable_tx_capture = 0;
 static unsigned int enable_ip_defrag = 0;
-static unsigned int transparent_mode = standard_linux_path;
+static unsigned int transparent_mode = driver2pf_ring_non_transparent;
+//static unsigned int transparent_mode = standard_linux_path;
 static u_int32_t ring_id_serial = 0;

@@ -546,4 +570,9 @@
         sprintf(buf + rlen, "Num Free Slots : %u\n",
                 get_num_ring_free_slots(pfr));
+#ifdef PKAP_ENABLED
```

```
+        rlen +=
+          sprintf(buf + rlen, "PKAP Ring?      : %d\n",
+                  pfr->pkap_ring?1:0);
+#endif
        } else
         rlen = sprintf(buf, "WARNING fsi == NULL\n");
@@ -1441,8 +1470,13 @@
    */

+
    if((!pfring_enabled) || (!pfr->ring_active))
      return (-1);

+#ifdef PKAP_ENABLED
+  atomic_inc(&pfr->num_ring_users);
+#else  /* PKAP */
    atomic_set(&pfr->num_ring_users, 1);
+#endif /* PKAP */

    /* [1] BPF Filtering (from af_packet.c) */
@@ -1468,5 +1502,14 @@
          skb->cloned);
 #endif
+
+
+#ifdef PKAP_ENABLED
+       atomic_dec(&pfr->num_ring_users);
+#else  /* PKAP */
        atomic_set(&pfr->num_ring_users, 0);
+#endif /* PKAP */
+
+
+
        return (-1);
      }
```

```
@@ -1735,5 +1778,11 @@
         if (free_parse_mem)
           free_parse_memory(parse_memory_buffer);
-        atomic_set(&pfr->num_ring_users, 0);
+
+#ifdef PKAP_ENABLED
+       atomic_dec(&pfr->num_ring_users);
+#else  /* PKAP */
+       atomic_set(&pfr->num_ring_users, 0);
+#endif /* PKAP */
+
         return (-1);
        }
@@ -1786,5 +1835,10 @@
    if (free_parse_mem)
      free_parse_memory(parse_memory_buffer);
-  atomic_set(&pfr->num_ring_users, 0);
+
+#ifdef PKAP_ENABLED
+       atomic_dec(&pfr->num_ring_users);
+#else  /* PKAP */
+       atomic_set(&pfr->num_ring_users, 0);
+#endif /* PKAP */

   return (0);
@@ -2075,6 +2129,11 @@
   hdr.len = hdr.caplen = skb->len + displ;

+
   /* Avoid the ring to be manipulated while playing with it */
   read_lock_bh(&ring_mgmt_lock);
+
+  /*
+   * BOOKMARK01: PKAP_ENABLED hook search
+   */
```

```
   /* [1] Check unclustered sockets */
@@ -2408,4 +2467,13 @@
     return -EPERM;

+#ifdef PKAP_ENABLED
+  pkap_debug("sock->type = %d\n",sock->type);
+  if (sock->type == (SOCK_RAW + 10)) {
+         sock->type = SOCK_RAW;
+         pkap_debug("Flag to create pkap ring structure...\n");
+         //FIXME:Start custom settings
+  }
+#endif
+
   if (sock->type != SOCK_RAW)
     return -ESOCKTNOSUPPORT;
@@ -2459,4 +2527,8 @@

   sk->sk_destruct = ring_sock_destruct;
+
+#ifdef PKAP_ENABLED
+  pfr->pkap_ring = 0; // Use setsockopt to turn on
+#endif

   ring_insert(sk);
@@ -3298,5 +3370,5 @@
   u_int16_t rule_id, rule_inactivity;
   packet_direction direction;
-
+  pkap_debug("Entering ring_setsockopt() with optname %d\n",optname);
   if (pfr == NULL)
     return (-EINVAL);
@@ -3415,4 +3487,5 @@

   case SO_SET_APPL_NAME:
```

```
+    pkap_debug("GOT INSIDE SET APPL_NAME\n");
     if (optlen >
        sizeof(applName) /* Names should not be too long */ )
@@ -3829,5 +3902,23 @@
     }
     break;
-
+#ifdef PKAP_ENABLED
+  case SO_SET_PKAP_RING:
+    if (pkap_pfr && (pkap_pfr->pkap_ring == 0)) {
+          pkap_debug("ERROR: Can only be one PKAP Ring!\n");
+          return -EINVAL;
+    } else if (pkap_pfr && (pkap_pfr->pkap_ring == 1)) {
+          pkap_debug("WARNING: This ring is already a PKAP Ring...\n");
+          ret = 0;
+    } else {
+          pkap_pfr = pfr;     /* This exports the current pfr to kernel
+                                 space so a PKAP-enable kernel thread can
+                                 read from this ring through an extern
+                                 declared pkap_pfr variable */
+          pkap_pfr->pkap_ring = 1;
+          pkap_debug("Converted PF_RING to PKAP_RING\n");
+          ret = 0;
+    }
+    break;
+#endif /* PKAP_ENABLED */
   default:
     found = 0;
```

Bibliography

1. Anderson, David P., et al. "A file system for continuous media," *ACM Trans. Comput. Syst.*, *10*(4):311–337 (1992).
2. Bar, Moshe. *Linux File Systems*. Osborne/McGraw-Hill, 2001.
3. Barton, Jim. "From Server Room to Living Room," *Queue*, *1*(5):20–32 (2003).
4. Benvenuti, Christian. *Understanding Linux Networking Internals*. O'Reilly Media, Inc., 2006.
5. Birch, Samuel, "PKAP 1.0 Source Code," Jan 2010.
6. Birrell, A D, et al. *The Echo distributed file system*. Technical Report, 1993.
7. Bishop, Matt. *Computer Security: Art and Science*. Addison Wesley, 2003.
8. Biswas, Amitava and Purnendu Sinha. "Efficient real-time Linux interface for PCI devices: A study on hardening a Network Intrusion Detection System." *The Fifth System Administration and Network Engineering Conference, SANE2006*. 2006.
9. Bovet, Daniel and Marco Cesati. *Understanding the Linux Kernel, Third Edition* (3 Edition). O'Reilly Media, Inc., November 2005.
10. Bovet, Daniel P. and Marco Cesati. *Essential Linux Device Drivers* (3rd Edition). Prentice Hall Open Source Software Development Series, Prentice Hall PTR, 2008.
11. Chang, Fay, et al. "Bigtable: a distributed storage system for structured data." *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. 205–218. Berkeley, CA, USA: USENIX Association, 2006.
12. Combs, Gerald, "Libpcap File Format." `http://wiki.wireshark.org/Development/LibpcapFileFormat/`, Dec 2008.
13. Cooperstein, Jerry. *Writing Linux Device Drivers: a guide with exercises*. Jerry Cooperstein, 2009.
14. Deri, Luca, "PF_RING Source Code." `http://www.ntop.org/download.html#PF_RING`, 2009. Access source directly through `svn co https://svn.ntop.org/svn/ntop/trunk/PF_RING/`.
15. Deri, Luca, et al. "Improving Passive Packet Capture: Beyond Device Polling." *In Proceedings of SANE 2004*. 2004.
16. Fenner, Bill, et al., "libpcap-1.0.0.tar.gz." `http://www.tcpdump.org/release/libpcap-1.0.0.tar.gz`, Oct 2008. Source Code Tarball.

17. Ghemawat, Sanjay, et al. "The Google file system," *SIGOPS Oper. Syst. Rev.*, *37*(5):29–43 (2003).

18. Giampaolo, Dominic. *Practical File System Design with the Be File System* (1 Edition). Morgan Kaufmann Publishers, 1999.

19. Gonzalez, Jose Maria and Vern Paxson. "pktd: A Packet Capture and Injection Daemon." *Passive and Active Measurement Workshop*. 2003.

20. Iannaccone, Gianluca, et al. "Monitoring very high speed links." *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*. 267–271. New York, NY, USA: ACM, 2001.

21. Kroah-Hartman, Greg. "Things you should never do in the kernel," *Linux J.*, *2005*(133):9 (2005).

22. Lamping, Ulf, et al., "Wireshark User's Guide." `http://www.wireshark.org/docs/`, 2008.

23. Larabel, Michael, "Real World Benchmarks Of The EXT4 File-System." `http://www.phoronix.com/scan.php?page=article&item=ext4_benchmarks`, Dec 2008.

24. Love, Robert. *Linux Kernel Development, Second Edition* (2 Edition). Novell Press, January 2005.

25. Mandia, Kevin, et al. *Incident Response & Computer Forensics, Second Edition* (2nd Edition). McGraw-Hill/Osborne, 2003.

26. Mauerer, Wolfgang. *Professional Linux Kernel Architecture*. Wrox Programmer to Programmer, Wiley Publishing, Inc., 2008.

27. McGrath, Roland, et al., "GNU C Library." `http://ftp.gnu.org/gnu/glibc/glibc-2.11.1.tar.bz2`, Dec 2009.

28. McVoy, L W and S R Kleiman. "Extent-like performance from a UNIX file system." *In Proceedings of the USENIX Winter Technical Conference*. 1991.

29. Ousterhout, J K, et al. "A Trace-Driven Analysis of the UNIX 4.2 BSD File System." *In Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP 85*. 15–24. 1985.

30. Ousterhout, John and Fred Douglis. *Beating the I/O Bottleneck: A Case for Log-Structured File Systems*. Technical Report, 1988.

31. Roesch, Martin, "The Story of Snort: Past, Present and Future." `http://www.net-security.org/article.php?id=860`, 2005. Recorded interview with Snort Creator.

32. Roesch, Martin, "DaemonLogger Web Site." `http://www.snort.org/users/roesch/Site/Daemonlogger/Daemonlogger.html`, Nov 2008.

33. Rosenblum, Mendel and John Ousterhout. “The LFS storage manager.” *In USENIX Summer*. 315–324. 1990.

34. Rosenblum, Mendel and John K. Ousterhout. “The design and implementation of a log-structured file system,” *ACM Trans. Comput. Syst.*, *10*(1):26–52 (1992).

35. Schneider, Fabian and Jörg Wallerich. “Performance evaluation of packet capturing systems for high-speed networks.” *CoNEXT ’05: Proceedings of the 2005 ACM conference on Emerging network experiment and technology*. 284–285. New York, NY, USA: ACM, 2005.

36. Seltzer, Margo Ilene. *File system performance and transaction support*. Technical Report, 1992.

37. Staelin, Carl Hudson. *High Performance File System Design*. Technical Report, 1991.

38. Torvalds, Linus, et al., “Linux Kernel Source Code.” `http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.30.10.tar.bz2`, Dec 2009.

39. Venkateswaran, Sreekrishnan. *Essential Linux Device Drivers*. Prentice Hall, 2008.

40. Zhang, Zhihui and Kanad Ghose. “hFS: a hybrid file system prototype for improving small file and metadata performance,” *SIGOPS Oper. Syst. Rev.*, *41*(3):175–187 (2007).

Vita

Samuel W. Birch graduated from Ebenezer Faith Christian School in Plymouth, Pennsylvania. He entered undergraduate studies at the United States Air Force Academy in Colorado where he graduated with a Bachelor of Science degree in May 1996. He received a Regular Commission upon graduation.

His first assignment was at Scott AFB as a Network Engineer in the 375th Communications Group in June 1996. In August 1998, he was assigned to the 375th Operations Group, Scott AFB, Illinois where he served as the executive officer to the Group Commander. During 1999, he was assigned to Kelly AFB where he served as a Network Security Engineer for the Air Force Information Operations Center. In 2001, he became the Chief Countermeasures Architect in the Information Operations Technology Division. In 2003, he separated from Active Duty to serve in the Air Force as a DoD Civilian—performing the roles of Chief Countermeasures Architect and the Information Operations Platform Lead Engineer. In 2008, he entered the Graduate School of Engineering and Management, Air Force Institute of Technology. Upon graduation, he will be assigned to the 688th Information Operations Wing, Lackland AFB, TX.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
05–03–2010	Master's Thesis	Jun 2008 — Mar 2010

4. TITLE AND SUBTITLE

Performance Characteristics of a
Kernel-Space Packet Capture Module

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Birch, Samuel W., IA-04, DAF

5d. PROJECT NUMBER

NO FUNDS PROVIDED

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCO/ENG/10-03

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

688th Information Operations Wing
Attn: Mr. Chet M. Wall
102 Hall Boulevard, Suite 310
San Antonio, TX 78243
(210)977-6603

10. SPONSOR/MONITOR'S ACRONYM(S)

90 IOS/TA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This research attempts to improve the efficiency of capturing network packets to disk using commodity, general-purpose hardware and operating systems. It examines the bottlenecks between NIC and disk, implements a kernel-space capture capability to improve storage efficiency, and analyzes the performance characteristics of this approach. Results show that a kernel-space NIC-to-Disk capture module is both possible and beneficial. The proof of concept PKAP kernel-space packet capture module can capture packets to disk with a packet drop rate 8.9% less than the user-space equivalent, at a 95% confidence interval. During the high levels of disk I/O contention produced by queries for the captured data, the PKAP implementation shows a 3% reduction in CPU utilization, and overall the PKAP implementation reduces memory utilization of the capture process by 16%.

15. SUBJECT TERMS

packet capture, kernel thread, ring buffer, network, security

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Robert F. Mills
U	U	U	UU	132	19b. TELEPHONE NUMBER *(include area code)* (937) 255–3636, ext 4527; Robert.Mills@afit.edu